Dispense prodotte dal LIFO (Laboratorio Informatico Free ed Open)

27 novembre 2004

Indice

Capitolo 1

Introduzione a Linux

di Carlo Pinciroli

1.1 Che cos'é Linux

Linux è un sistema operativo derivato da UNIX, e perciò conserva molte delle caratteristiche di quest'ultimo, tra cui le più importanti sono:

La multiutenza: più utenti possono accedere al sistema (che può essere formato da un terminale o da una rete di terminali) indipendentemente l'uno dall'altro; in altri termini l'attività di un utente non può compromettere l'attività di un altro utente (ad esempio l'utente "gianni" non può cancellare i dati di "pinotto" né modificarli, senza il consenso di "pinotto"). Per garantire tutto questo, il sistema operativo prevede dei veri e propri diritti (di lettura, scrittura ed esecuzione) da applicare a file e directory: in questo modo chi non possiede i diritti necessari non può intervenire sul file o sulla directory che vorrebbe. Nessun utente normale può modificare i file di configurazione e di avvio del sistema, per evidenti ragioni di sicurezza: è dunque necessaria l'esistenza di un utente speciale, denominato root (in inglese "radice"), che ha diritti illimitati sul sitema: l'utente root funge quindi da amministratore. Una conseguenza diretta di tutto ciò è la necessità di identificare l'utente al lavoro, al fine di definire i suoi diritti sul sistema: questa identificazione viene effettuata all'atto del login, senza il quale non si può avere accesso alle risorse del sistema. Durante il login vengono richiesti nome utente e password.

Il multitasking: da task, in inglese "compito". È la possibilità di eseguire più programmi contemporaneamente, senza che un programma possa influenzarne un altro: il blocco di un prorgamma in Linux (e in UNIX, naturalmente) non pregiudica il funzionamento di tutto il sistema come di norma accade in Windows quando appaiono le famose schermate blu. Per questo Linux è considerato un sistema stabile.

Nonostante storicamente Linux interagisse con l'utente per lo più attraverso la console (l'analogo del "Prompt di MS-DOS" di Windows), esiste anche un ambiente grafico, simile a quello Microsoft, denominato X Window. Può avere diversi aspetti grafici e diversi modi di essere utilizzato, perchè diversi sono i gestori delle finestre (Window Manager in inglese), tra i quali spiccano per fama e semplicità d'uso KDE e GNOME.

Una delle più famose e, al tempo stesso, positive caratteristiche di Linux è la libertà di distribuire i sorgenti del sistema operativo (ovvero il sistema operativo non ancora eseguibile, quindi modificabile a piacimento dall'utente al fine di porre migliorie o correggere eventuali errori), e quindi di copiare sul proprio computer Linux semplicemente scaricandolo da Internet. Dal momento però che la grandezza media di un download di questo tipo è piuttosto consistente, e che la scelta delle parti da scaricare e poi installare è vasta e piuttosto difficoltosa, alcune aziende hanno pensato di fornire su CD-ROM il sistema operativo già pronto da installare con le parti del sistema necessarie, i programmi di uso più comune e una procedura di installazione e configurazione semplificata. Queste distribuzioni su CD-ROM (i cui nomi sono, per citarne alcune: RedHat, Mandrake, SuSe, Debian, Slackware...) sono facilmente reperibili, a bassissimo prezzo (spesso meno di 5 euro) e differiscono fra di loro per l'hardware supportato, la scelta dei programmi e la posizione di certi file di configurazione del sistema.

1.2 Concetti di base sul funzionamento di Linux

Il filesystem è il complesso dei meccanismi che gestiscono file e directory del sistema: attraverso di esso, è possibile creare, modificare, cancellare e spostare file e directory. Cosìcome Windows possiede un filesystem, anche Linux possiede il proprio. In Linux però il concetto di file è piuttosto particolare, perché è molto più astratto che in Windows. Infatti si usa dire che in Linux "tutto è un file", ovvero qualunque parte fisica o logica del computer (che sia un disco, una scheda sonora o di rete, un modem, uno schermo, ecc.) viene gestita dal sistema come se fosse un insieme di file. Ad esempio, per stampare un documento, in effetti Linux scrive su un file che rappresenta la stampante; è poi il "sottobosco" del sistema che si occupa di tradurre questa brutale scrittura in segnali comprensibili alla periferica. In altre parole, Linux vede tutto come se fosse un file. Per poter gestire una periferica, bisogna quindi far sìche l'insieme dei file relativi alla periferica stessa faccia parte della struttura delle directory di Linux: questa operazione di aggiunta di file a quelli di Linux è detta montaggio (in inglese mounting). Quindi, se una cosa non è montata, Linux non la vede, e quindi non può essere utilizzata. Un esempio evidente di tutto questo si ottiene quando si cerca di tradurre un indirizzo di un file sotto Windows (ad es. `C:\WINDOWS\WIN.INI`) in uno per Linux. Tanto per cominciare, sotto Linux non esiste l'espressione A: o C:, perché le varie unità a disco fanno parte della struttura delle directory, se sono già montate. Il montaggio di un'unità a disco fa sìche l'intero contenuto del disco appaia come una sottodirectory, quindi ad esempio se l'hard disk C: è stato montato nella directory `/mnt/winc`, l'indirizzo in formato Linux sarà `/mnt/winc/windows/win.ini` Attenzione: Linux è case sensitive quando tratta i nomi di file, quindi `WIN.INI` e `win.ini` sono due file differenti.

La struttura delle directory di Linux è abbastanza standard (alcune sottodirectory cambiano da distribuzione a distribuzione)

- `/home` : qui vengono messi i file personali degli utenti. Ogni utente ha una sua sottodirectory in cui può fare ciò che vuole: ad esempio "gianni" ha `/home/gianni` e "pinotto" ha `/home/pinotto`. L'analogo Windows è più o meno `C:\Documenti`.
- `/root` : la directory personale di "root"
- `/etc` : contiene molti file di configurazione del sistema e di vari programmi. Un pò come `C:\Windows\Command` o `C:\Windows\System`.
- `/usr` : contiene i file dei programmi installati, i manuali e molto altro. Più o meno come la `C:\Programmi`.
- `/dev` : contiene tutti i file che rappresentano una periferica, ad esempio
  - `/dev/hda` è l'hard disk primario (il C: di windows, di solito)
  - `/dev/hdb` è il drive secondario, slave di `/dev/hda` (quindi o un hard disk o un cdrom)
  - `/dev/hda1` indica la prima partizione di C:
  - `/dev/hda3` è la terza partizione di C:
  - `/dev/ttsy0` è la porta COM1
  - `/dev/ttsy1` è la porta COM2

  e così via.

1.3 Introduzione, struttura di base di X e avvio

X è il più famoso sistema grafico per Unix, e quindi anche per Linux. Sono sinonimi i termini X, X Window e X Window System, mentre la dicitura "X Windows" è errata.

La struttura di X è organizzata a strati, di cui il più basso rappresenta l'hardware del computer e il più alto le applicazioni. L'hardware (di solito solo tastiera + mouse + scheda video + schermo) è gestito da X attraverso un SERVER, ovvero un "servitore", che ha il compito di offrire le funzionalità grafiche utilizzate poi dai CLIENT. I client (in italiano clienti) sono infatti programmi che usano l'ambiente grafico comunicando con il server X e facendo uso di librerie chiamate XLIB. Al fine di garantire la comunicazione fra server e client sono state predisposte delle regole di linguaggio, raccolte in un protocollo detto con scarsa fantasia PROTOCOLLO X. Il rapporto esistente fra server e client di X ricalca quello che sussiste in ambiente di rete, ed infatti il server X fra le funzionalità garantite ha anche quella di permettere una connessione da remoto. Per chiarire con un'immagine il rapporto che sussiste fra server e client, si può pensare che il server sia un venditore che tiene aperto il proprio negozio in attesa di clienti; che i clienti si presentino o meno, il negozio rimane aperto, ed appena un cliente si presenta viene subito servito se è possibile farlo. Il protocollo è in un certo senso la "lingua" (italiano, inglese...) che viene utilizzata per comunicare dal negoziante e dal cliente. Una collezione di serventi grafici molto famosa è XFree86: ogni servente grafico è specifico per un tipo di scheda video, ma non mancano serventi grafici per schede video generiche. I serventi grafici possono quindi essere considerati come delle specie di "driver",

per dirla con una similitudine con Windows, ma come abbiamo visto i server fanno molto di più di quello che fanno i driver in ambiente Microsoft. I client sono invece i programmi, che siano un gioco, Microsoft Office, o altro.

Tra i programmi clienti fondamentali, una particolare menzione merita il gestore delle finestre (Window Manager). È un programma che mette a disposizione le funzionalità che l'utente necessita per poter usare il sistema grafico, quindi ad esempio una barra come la "Avvio" di Windows, le caselline per ridimensionare/spostare/chiudere le finestre, e oggetti simili. Tra i più famosi Window Manager ricordiamo KDE, Gnome, fvwm con le varianti fvwm2, e fvwm95-2, twm, Afterstep e IceWM. È importante però sapere che è possibile utilizzare X anche senza un gestore delle finestre, anche se questo fa parte di argomenti che esulano dalla trattazione attuale.

Una delle differenze più sorprendenti che X ha nei confronti di Windows è data dal fatto che in X esistono degli schermi virtuali. Anche se lo schermo fisico (quello che si sta guardando) è uno soltanto, X permette di gestire più schermi e di passare da uno schermo ad un altro con la stessa facilità con cui si passa da un programma ad un altro. La quantità di schermi virtuali disponibili dipende dalle impostazioni che si sono adottate. Un'altra differenza con Windows, ma meno marcata, è la necessità di un mouse a tre tasti per X, quando invece Windows richiede solo un mouse a due tasti. Molte operazioni (fra cui quelle di copia-incolla) sono spesso legate al tasto centrale del mouse, e quindi non averlo costringe a configurare X in modo particolare, al fine di emulare il tasto centrale mancante tramite la pressione contemporanea dei due tasti del mouse.

È possibile impostare il boot di linux per avviare X all'avvio, nel qual caso il login che si presenta è di tipo grafico. Altrimenti, se si esegue del lavoro in ambiente testuale (console), l'avvio di X si ottiene col comando startx.

1.4 Programmi di uso comune

La quantità di applicativi per X è considerevole, e non è possibile né in questa sede né in futuro coprirne l'intera lista. Ci limiteremo a fornire le possibili alternative ai più famosi programmi per Windows offerti da X per Linux con KDE come window manager. Naturalmente anche Gnome o gli altri window manager offrono applicativi analoghi, alcuni dei quali sono comuni a tutti i window manager in quanto vengono forniti insieme a XFree86. Almeno per quanto riguarda KDE, l'utilizzo dei programmi è molto simile allo stile di Windows. Unica vera differenza, in Linux non esiste il "doppio clic".

1.4.1 Avvio di programmi

Le modalità sono le stesse di Windows: si può utlizzare la barra "K" oppure le icone sul desktop, ma senza fare uso del doppio clic!

1.4.2 Operazioni sulle finestre

Sulla barra del titolo a destra ci sono tre icone, che come in Windows permettono rispettivamente di ridurre a icona, massimizzare e chiudere la finestra. A sinistra del titolo invece troviamo prima l'icona del programma a cui la finestra si riferisce: cliccando su di esso compare un menù che permette di fare le stesse

operazioni delle icone a destra. L'altra icona tra quella menzionata e il testo del titolo, se cliccata, fa sì che quella finestra compaia in tutti gli schermi virtuali.

1.4.3 Personalizzazione dell'ambiente

È possibile effettuarla tramite il KDE Control Center, navigando fra i menù a tendina.

1.4.4 Operazioni sui file

Konqueror può essere tranquillamente paragonato a "Esplora risorse" (o "Gestione risorse") di Windows e ne ricalca le modalità di utilizzo. Per un analogo di Winzip ci si può rivolgere a GnoZip. xterm è un terminale testuale emulato. Quando si parlerà dei comandi della console, si vedrà l'utlità di poter accedere ad un terminale virtuale. Al momento ci limitiamo a paragonarlo al "Prompt di MS-DOS" di Windows. Una delle comodità di Windows in fase di installazione è data da InstallShield, e anche X offre delle funzionalità analoghe (ma non identiche),anche se diverse da distribuzione a distribuzione: nella RedHat e nella Mandrake sono offerti gli archivi RPM che si possono gestire con kpackage.

1.4.5 Lavoro d'ufficio

Se quello di cui si ha bisogno è Office, Linux offre i notevoli OpenOffice.org e KOffice. Il primo dei due è fra l'altro identico al sosia sotto Windows ed è completamente compatibile con esso. Anche le modalità di utilizzo e i nomi delle opzioni sono praticamente gli stessi.

1.4.6 Internet

L'Accesso remoto è "tradotto" in KPPP sotto Linux, il gestore della posta Outlook in KMail e i browser disponibili sono Netscape/Mozilla, Opera (disponibili sia per Windows che per Linux), Konqueror (come Explorer, per KDE) e Galeon (per Gnome).

1.4.7 Multimedia

Per quanto riguarda la musica e gli mp3, xmms è il parente Unix di Winamp; la grafica è invece maneggiabile con programmi come gimp (fotoritocco), gqview, blender (grafica 3d); anche OpenOffice.org offre programmi di questo genere.

1.5 Conclusioni

Per concludere, non è difficile convincersi che praticamente ogni cosa si possa fare con un programma a pagamento per Windows sia tranquillamente realizzabile in Linux con programmi Open Source. L'unica (relativa) falla di Linux è la scarsità di giochi, se paragonata alla considerevole quantità di videogames disponibili per la piattaforma Microsoft.

1.6 Cenni generali sul concetto di shell

Il programma più importante in un sistema operativo, dopo il kernel, è l'ambiente in cui è possibile interagire con l'hardware del computer. In realtà il dialogo con l'hardware è già garantito dal kernel, e a questo è dovuta la sua fondamentale importanza; quello che rimane all'utente è quindi il compito di impartire comandi al kernel. Questa attività è piuttosto complessa, e quindi si usa racchiudere il kernel stesso in una "conchiglia" (in inglese shell) con la quale l'utente interagisce effettivamente. In altre parole, la shell è l'ambiente che permette all'utente di interagire con il kernel mettendo a disposizione comandi comprensibili dall'uomo e interpretabili dal kernel. Una volta interpretati, i comandi vengono mandati all'hardware che si occupa di eseguirli. Una shell dunque può essere un sistema grafico, come Windows o X, oppure un ambiente testuale come DOS o la console di Linux; concettualmente non ci sono differenze fra ambiente grafico e testuale in termini di possibilità di interazione con il sistema: entrambi gli ambienti permettono un'efficace utilizzo della macchina, anche se la difficoltà di utilizzo dell'ambiente testuale e la sua versatilità sono spesso molto maggiori.

1.7 La shell Bash

Bash è la shell standard di molti sistemi Unix fra cui anche Linux; è l'evoluzione della shell scritta da Stephen Bourne per Unix, chiamata appunto shell Bourne. Bash significa dunque Bourne Again SHell (in inglese "ancora shell Bourne"), ed associa le funzioni della shell Bourne alle funzioni di altre due shell piuttosto famose, la shell Korn (ksh) e la shell C (csh).

L'eseguibile della shell Bash è `/bin/bash` ed è paragonabile a `COMMAND.COM` degli ambienti Microsoft.

Può essere avviato in due modalità:

- la modalità interattiva, che permette all'utente di inserire comandi. Viene quindi visualizzato un invito a digitare detto "prompt dei comandi".
- la modalità non interattiva, in cui vengono eseguiti gli script di shell (come i file batch di DOS) e i comandi esterni; gli script verranno trattati più avanti. In questa modalità non viene visualizzato alcun prompt.

La shell in modalità interattiva, dunque, è ciò che si presenta quando abbiamo appena finito la procedura di login; sullo schermo appare un messaggio del tipo

```
[pippo@localhost etc]$ _
```

in cui il prompt è formato da tutti i caratteri fino a $ compreso, mentre il trattino è il cursore lampeggiante che indica dove si sta per scrivere; la linea visualizzata è quindi una "riga di comando". Il carattere $ si presenta quando l'utente al lavoro è un utente normale; se invece si sta lavorando come root, il carattere visualizzato è #.

Non appena avviata, Bash legge i file di configurazione `/etc/profile` e `/.bash_profile` (se quest'ultimo non esiste legge `/.bash_login`; nel caso nemmeno questo esista, prova con `/.profile`; altrimenti rinuncia alla lettura). I

file menzionati contengono le impostazioni dell'ambiente di lavoro; ne parleremo più diffusamente tra qualche pagina, una volta introdotte le variabili.

1.7.1 Inserimento ed esecuzione dei comandi

Un comando altro non è che una serie di parole da scrivere al prompt; la pressione del tasto Invio fa sìche la shell legga quanto è stato inserito e lo intepreti; una volta intepretato, il comando viene eseguito.

I comandi che vengono eseguiti sono raccolti in una lista ordinata cronologicamente (detta "history dei comandi") che permette di ritrovarli facilmente e di non doverli cosìriscrivere.

Alcuni comandi sono forniti dalla shell stessa, e allora sono detti "comandi interni". Ma la maggior parte dei comandi disponibili non è interna, bensìè un programma esterno alla shell che spesso risiede nella directory `/bin` o `/sbin`. Quando si eseguono i comandi esterni, Bash carica una nuova shell Bash in modo non interattivo ed è questa shell figlia ad eseguire il comando esterno.

La shell prevede dei tasti speciali per eseguire particolari e comode funzioni:

- Tab: premendo Tab, la shell cerca di completare quanto è stato scritto secondo lo schema seguente:
  - se la parola inizia con ~ , la interpreta come un nome di un utente e cerca un utente che inizi con le lettere digitate; se lo trova completa la parola col nome trovato e la visualizza sulla riga di comando;
  - se la parola inizia con $, la interpreta come un nome di una variabile e si comporta come sopra;
  - altrimenti interpreta la parola come un nome di file, e completa seguendo le regole succitate.
- Freccia su: richiama sulla linea l'ultimo comando digitato; premendo ancora freccia su, compare il penultimo e cosìvia.
- Freccia giù: richiama il comando successivo nella lista a quello visualizzato.
- Freccia a destra / sinistra: permette di spostarsi a destra/sinistra tra i caratteri del comando attivo al fine di correggerlo più agevolmente.
- Ins: fa passare dalla modalità di inserimento a quella di sovrascrittura.
- Inizio / Fine: sposta il cursore all'inizio/fine della riga di comando

Il modo più semplice di eseguire un comando è di scrivere il suo nome e premere Invio, ad esempio:

```
[pippo@localhost etc]$ pwd [-> Invio]
/home/pippo
```

questo comando visualizza il nome completo della directory in cui si sta lavorando al momento (pwd significa print working directory).

1.7.2 Parametri

pwd non richiede particolari informazioni per poter svolgere il proprio lavoro, quindi è possibile eseguirlo nel modo visto. Ma ci sono comandi che invece necessitano di informazioni aggiuntive che l'utente può specificare facendo seguire al nome del comando una lista di altre stringhe (dette parametri o argomenti) separate l'una dall'altra da spazi. Ad esempio il comando rm (remove) cancella i file; eseguendo tale comando senza specificare argomenti:

```
[pippo@localhost etc]$ rm [-> Invio]
rm: too few arguments
```

si riceve un messaggio di errore che esprime la necessità di ulteriori informazioni. Il modo corretto di eseguirlo è

```
[pippo@localhost etc]$ rm pippo.txt [-> Invio]
```

in questo modo si cancella il file pippo.txt dalla directory corrente. Una cosa da sapere è che un file, una volta cancellato, non è più recuperabile; il vecchio DOS offriva l'utility UNDELETE che permetteva di recuperare in qualche caso i file rimossi, ma su Linux la cancellazione non prevede questa possibilità. Una ragione in più per lavorare come utente normale e non come root, è appunto la certezza di non cancellare file importanti senza i quali il sistema sarebbe compromesso.

In generale, quindi, la struttura di un comando è la seguente:

```
nomecomando parametro1 parametro2 ... parametroN
```

Abbiamo visto che un parametro può essere un nome di un file, ma i parametri possono essere utilizzati per specificare quale opzione di un comando si vuole utilizzare. Ad esempio, rm prevede un'opzione per attivare la richiesta di conferma di rimozione di un file al fine di prevenire cancellazioni indesiderate. Il comando con l'opzione appare così:

```
[pippo@localhost etc]$ rm -i pippo.txt [-> Invio]
rm: remove 'pippo.txt' ? _
```

l'opzione -i (abbreviazione di interactive) fa sìche compaia un messaggio di conferma che aspetta una y o una n come risposta per procedere o annullare l'operazione.

Dall'esempio visto, si notano due fatti:

- le opzioni precedono i nomi di file
- le opzioni si distinguono dai file per la presenza di un `-` iniziale

nessuno dei due fatti menzionati è però una regola fissa, si tratta soltanto di una convenzione.

`rm` prevede anche un'opzione `-r` che permette di cancellare i file specificati nella directory corrente e anche nelle sottodirectory. Volendo eseguire `rm` con entrambe le opzioni `-i` e `-r` è possibile raggrupparle come segue:

```
[pippo@localhost etc]$ rm -ir pippo.txt [-> Invio]
rm: remove 'pippo.txt' ? _
```

Poniamo ora il caso di essere nella directory `/home/pippo`, e di voler spostare il file `trippa.gnam` che si trova nella directory `/home/pippo/tantafame`, per metterlo nella directory corrente. Un modo per farlo è il seguente:

```
[pippo@localhost pippo]$ mv /home/pippo/tantafame/trippa.gnam \
/home/pippo [-> Invio]
```

il comando `mv` (move) sposta il file specificato come primo argomento nella directory specificata come secondo argomento. La sintassi usata è un po' pesante, perchè abbiamo specificato l'intero percorso del file `trippa.gnam` e della directory corrente. Ma se ci troviamo in `/home/pippo`, è possibile sottintendere questa parte del percorso, scrivendo:

```
[pippo@localhost pippo]$ mv tantafame/trippa.gnam /home/pippo \
[-> Invio]
```

è da notare che il primo argomento non comincia per /, quindi viene interpretato da Bash come una sottodirectory di `/home/pippo`. Sarebbe comodo poter sottintendere anche `/home/pippo` nel secondo argomento, però una scrittura del tipo

```
[pippo@localhost pippo]$ mv /home/pippo/tantafame/trippa.gnam \
[-> Invio]
```

genera un errore in quanto mv vuole che la destinazione dello spostamento sia specificata. Allora si può utilizzare il carattere . che è il nome di un file che rappresenta la directory corrente. Ovvero, si può scrivere

```
[pippo@localhost pippo]$ mv /home/pippo/tantafame/trippa.gnam . \
[-> Invio]
```

e stavolta tutto funziona, perché viene intepretato da mv nel modo esatto. Se . è la directory corrente, .. è invece un file che punta alla directory madre di quella corrente. Nel nostro esempio, il comando

```
[pippo@localhost pippo]$ mv /home/pippo/tantafame/trippa.gnam \
.. [-> Invio]
```

sposta il file trippa.gnam dalla directory `/home/pippo/tantafame` alla directory `/home`. mv è usato anche per cambiare nome ai file (come il comando REN di DOS):

```
[pippo@localhost tantafame]$ mv trippa.gnam cosciotto.slurp \
[-> Invio]
```

questo comando, eseguito nella directory `/home/pippo/tantafame`, cambia nome al file `trippa.gnam` facendolo diventare `cosciotto.slurp`.

1.7.3 Caratteri speciali

Nel caso in cui si volessero cancellare tutti i file di una directory, con le conoscenze attuali dovremmo eseguire rm specificando come parametri i nomi di tutti i file; la scomodità della procedura è evidente, e per questo la shell Bash prevede dei caratteri speciali che rendono molto più agevole la selezione di liste di file. I

caratteri speciali sono i seguenti: * (indica un insieme qualunque di caratteri), ? (è un segnaposto per un carattere qualunque), [...] (definisce un insieme di caratteri) e ~ (è il nome della directory home).

Alcuni esempi chiariranno l'uso dei caratteri speciali. Supponiamo che la directory corrente contenga i file `pippo.txt`, `pluto.txt`, `paperino.jpg`, `lippo.dat` e `gippo.gif`. Per cancellare tutti i file che cominciano per p si può scrivere

```
[pippo@localhost etc]$ rm p* [-> Invio]
```

La shell Bash interpreta p* come p seguito da una qualunque stringa, e i file `pippo.txt`, `pluto.txt` e `paperino.jpg` rispondono a questi requisiti. Quindi la shell, prima di mandare il comando in esecuzione, sostituisce p* con i nomi dei tre file indicati, trasformando il comando in

```
rm pippo.txt pluto.txt paperino.jpg
```

Il comando è ora pronto per essere eseguito. La sostituzione dei caratteri speciali non viene notificata dalla shell, cioé è fatta senza visualizzare alcun messaggio, quindi è bene controllare con attenzione che la stringa inserita generi la giusta lista di file.

Per cancellare tutti i file di una directory, procedere come segue:

```
[pippo@localhost etc]$ rm * [-> Invio]
```

qui * sta ad indicare una stringa qualsiasi, quindi qualunque nome di file va bene e allora vengono cancellati tutti. Se si vuol cancellare i file `gippo.gif`, `pippo.txt` e `lippo.dat`, si può eseguire il comando

```
[pippo@localhost etc]$ rm ?ippo* [-> Invio]
```

in cui il carattere ? viene interpretato dalla shell come un qualsiasi carattere. Se invece si vuol cancellare solo `gippo.gif` e `lippo.dat`, ma non `pippo.txt`, si può utilizzare la forma

```
[pippo@localhost etc]$ rm [g,l]ippo* [-> Invio]
```

in cui [g,l] indica che la shell deve scegliere solo fra g e l la sostituzione di carattere. Sono possibili selezioni multiple come [a-f,z] che indica l'insieme dei caratteri a, b, c, d, e, f, z.

Infine veniamo al carattere ~: se è da solo viene sostituito dalla shell con il nome completo della directory home, oppure se è seguito da caratteri viene sostituito con la home directory dell'utente il cui nome è quello che segue ~. Ad esempio, se in un sistema ci sono due utenti chiamati gianni e pinotto, e l'utente al lavoro è gianni, allora ~ viene sostituito con `/home/gianni` e ~pinotto viene sostituito con `/home/pinotto`.

1.7.4 Protezione dei caratteri speciali

Può capitare di dover lavorare su un file il cui nome contiene dei caratteri speciali, ad esempio `gi*gi`, ma come sappiamo il carattere * viene trasformato da Bash. In realtà, la trasformazione va a buon fine solo se esistono file che cominciano e finiscono per "gi"; se nessun file risponde a questi requisiti, Bash rinuncia alla trasformazione e lascia la stringa gi*gi invariata. Perciò, se l'unico

file che risponde ai requisiti di cominciare e finire per "gi" è `gi*gi`, non ci sono equivoci: il file a cui ci si riferisce è quello desiderato. Ma nel caso in cui in una directory si trovino i file `gi*gi` e `giangi`, il comando

```
[pippo@localhost etc]$ rm gi*gi [-> Invio]
```

cancellerà solo `giangi`, perché la sostituzione di * è andata a buon fine. Una volta cancellato questo file l'equivoco scompare, ma se ogni volta che si presenta una situazione ambigua l'unica soluzione fosse di fare piazza pulita dei file che "coprono" `gi*gi`, la comodità della shell verrebbe meno.

Per risolvere le ambiguità che coinvolgono i caratteri speciali è possibile "proteggerli" dalla trasformazione tramite un altro carattere speciale (sembra quasi un circolo vizioso, ma poi le cose funzionano), ovvero il carattere *backslash*. Per proteggere * basta quindi farlo precedere da *backslash* in questa maniera:

```
[pippo@localhost etc]$ rm gi\*gi [-> Invio]
```

In questo caso, la shell Bash non trasforma * ma lo lascia invariato, ed elimina la *backslash*; il comando diventa dunque

```
rm gi*gi
```

proprio come volevamo. E per cancellare un file di nome `gi\gi`? Basta proteggere \ facendolo precedere da un altro \, in questo modo:

```
[pippo@localhost etc]$ rm gi\\gi [-> Invio]
```

e l'intepretazione che Bash ne dà è

```
rm gi\gi
```

A ben guardare, anche il carattere " " (spazio) è speciale per la shell Bash: è infatti il carattere che separa fra loro gli argomenti di un comando. Può capitare però di dover lavorare con file il cui nome contiene uno spazio, ma naturalmente non si può digitare

```
[pippo@localhost etc]$ rm ciao ciao come stai io bene [-> Invio]
```

sperando che serva a cancellare il file `ciao ciao come stai io bene`. In questo caso la protezione che si può fare è la seguente:

```
[pippo@localhost etc]$ rm ciao\ ciao\ come\ stai\ io\ bene \
[-> Invio]
```

oppure, più intuitivamente e comodamente

```
[pippo@localhost etc]$ rm "ciao ciao come stai io bene" \
[-> Invio]
```

Nel caso in cui, infine, il file da cancellare fosse '`mi chiamo "giangi"`', il comando opportuno è

```
[pippo@localhost etc]$ rm "mi chiamo \"giangi\"" [-> Invio]
```

Se un comando è troppo lungo per stare su un'unica linea, è possibile andare alla linea successiva digitando *backslash* seguito da Invio. Questa procedura non dovrebbe stupire: il carattere *backslash* toglie ai caratteri speciali il loro significato speciale: quindi anche il tasto Invio, se preceduto da *backslash* viene privato della funzione di marcatore di fine comando. Un esempio:

```
[pippo@localhost etc]$ rm nelmezzodelcammin\ [-> Invio]
dinostravita [-> Invio]
```

questo cancella il file `nelmezzodelcammindinostravita`.

1.7.5 Variabili di shell e di ambiente

Alcune informazioni che caratterizzano la sessione di lavoro (come la directory corrente, l'utente al lavoro, ...) sono immagazzinate in memoria sotto forma di variabili. Una variabile è una specie di "scatola" nella quale è possibile conservare un valore, sia numerico che testuale. Per distinguere fra loro le "scatole", ovvero le variabili, si è pensato di dotarle di un nome. Le variabili di shell hanno validità solo nella shell con cui si sta lavorando. Ma sappiamo che quando si esegue uno script o un comando esterno viene aperta una nuova shell in modo non interattivo; perciò questa nuova shell non vedrà le variabili della shell madre. Però spesso si vuole che le variabili della shell su cui si lavora siano disponibili anche alle altre shell figlie aperte successivamente; allora è necessario trasformarle in variabili di ambiente.

Le variabili di shell più importanti sono:

- `PATH`: questa variabile contiene una lista di directory separate da : e serve alla shell per sapere dove trovare i comandi digitati. Quando si digita un comando, se il comando è interno l'esecuzione non richiede la lettura di `PATH`, ma se è esterno, la shell vuole sapere dove andare a cercarlo; allora legge la variabile PATH e prova ordinatamente a trovare il comando nella prima directory della lista, poi se non lo trova prova nella seconda e così via. Se il comando non viene trovato in nessuna delle directory, allora la shell emette un messaggio di errore (command not found). `PATH` di Linux è quindi come `PATH` di DOS.

- `HOME`: contiene l'indirizzo della directory home dell'utente al lavoro (`/home/gennaro` ad esempio)

- `PS1`: contiene il formato del prompt dei comandi, ovvero dell'invito a digitare che di solito appare nella forma `[pippo@localhost etc]$` Modificando questa variabile, cambia l'aspetto del prompt; è simile alla variabile `PROMPT` di MS-DOS.

- `HISTSIZE`: il numero massimo di comandi da conservare nella history (di solito 500 o 1000)

- `HISTFILE`: il file in cui scrivere la history (di solito `/.bash_history`)

- `PWD`: la directory corrente

- `OLDPWD`: l'ultima directory visitata prima di andare in quella attuale

- `BASH`: il percorso dell'eseguibile della shell Bash, ovvero `/bin/bash`

- `LOGNAME`: il nome con cui si è eseguito il login

- `USER`: il nome dell'utente al lavoro

Per dichiarare una variabile di shell di nome `PIPPO` e dargli un contenuto iniziale (ovvero in gergo inizializzarla), digitare

```
[pippo@localhost pippo]$ PIPPO="uncane" [-> Invio]
```

Per farla diventare un variabile di ambiente si utilizza il comando export:

```
[pippo@localhost pippo]$ export PIPPO [-> Invio]
```

Ma si può fare tutto in una volta, anziché in due passaggi:

```
[pippo@localhost pippo]$ export PIPPO="uncane" [-> Invio]
```

Per visualizzare il contenuto di una variabile, usare il comando echo in questo modo:

```
[pippo@localhost pippo]$ echo $PIPPO [-> Invio]
uncane
```

come si nota, per far capire alla shell che PIPPO non è il nome di un file ma il nome di una variabile si fa precedere PIPPO da un $.

Se sulla riga di comando scriviamo

```
[pippo@localhost pippo]$ echo $PI [-> Tab]
```

seguito dalla pressione del tasto Tab, la shell completa la riga visualizzando

```
[pippo@localhost pippo]$ echo $PIPPO
```

cioè proprio come fa coi nomi di file.

Per consultare la lista di tutte le variabili (che siano di shell o di ambiente) si può usare il comando set senza parametri, e per eliminare una variabile si deve digitare

```
[pippo@localhost pippo]$ unset PIPPO
```

Adesso possiamo riprendere brevemente il discorso sul contenuto di `/etc/profile` e `/.bash_profile`: questi file contengono una serie di dichiarazioni di variabili di ambiente e alcuni comandi per configurare l'ambiente di lavoro. `/etc/profile` ha una configurazione che si applica a tutti gli utenti; `/.bash_profile` invece contiene la configurazione personalizzata dell'utente.

1.7.6 Alias

Chi utilizza il DOS, per visualizzare la lista dei file di una directory utilizza il comando dir; su Linux, questo comando non si chiama dir ma ls, e per chi è abituato a digitare dir questo può essere uno spiacevole inconveniente. È allora possibile definire un nuovo nome (detto alias) per il comando ls col comando alias:

```
[pippo@localhost etc]$ alias dir="ls -l" [-> Invio]
```

l'opzione -l (long listing) di ls fa sì che la lista visualizzata sia provvista di molti dettagli sui file, più o meno come la lista presentata da dir. Quando impartiamo il comando dir alla shell, la prima cosa che viene fatta è la sostituzione di dir con ls -l, e poi vengono eseguite tutte le sostituzioni dei caratteri speciali. Ovvero:

```
[pippo@localhost etc]$ dir p* [-> Invio]
```

diventa

```
ls -l pippo.jpg pluto.gif
```

Per eliminare un alias, usare il comando unalias:

```
[pippo@localhost etc]$ unalias dir [-> Invio]
```

Se si vuole far sì che un alias sia impostato automaticamente dopo il login di Linux, bisogna semplicemente aggiungere il comando relativo nel file `/.bash_profile`; non è consigliabile aggiungerlo al file `/etc/profile` perché quest'ultimo contiene i dettagli di configurazione validi per tutti gli utenti.

1.7.7 Pipe e redirezione

Da quello che abbiamo visto finora, possiamo dire che ogni comando ha questa struttura concettuale di base:

- riceve dell'input (attraverso i parametri o attraverso domande effettuate in fase di esecuzione, come la richiesta di conferma di rm -i)
- emette dell'output (ad esempio presenta la lista dei file di una directory o mostra dei messaggi di errore)

Di solito, dunque, la tastiera è l'input e lo schermo è l'output; in gergo informatico, potremmo dire che la tastiera è lo standard input e lo schermo è lo standard output. Sappiamo che su Linux "tutto è un file", quindi non dovremmo stupirci di trovare dei file associati proprio allo standard input e allo standard output: e infatti, guardando nella directory `/dev` troviamo i tre file `/dev/stdin`, `/dev/stdout` e `/dev/stderr`.

I primi due sono proprio i file che si riferiscono allo standard input e allo standard output; il terzo invece è lo standard error. Per convenzione, i comandi hanno due tipi di output: quello normale che passa attraverso lo standard output, e quello relativo ai messaggi di errore che passa invece per lo standard error.

Quando viene eseguito, dunque, un comando legge caratteri dal file stdin, e scrive caratteri sul file stdout o stderr; ma nulla vieta di dirottare il flusso di caratteri verso il file stdout e di dirigerlo ad un file che vogliamo noi, perché sempre di file si tratta. È una procedura che almeno in teoria non dovrebbe generare errori.

L'operazione di dirottare il flusso di caratteri in entrata o in uscita da un comando si chiama appunto redirezione, e si effettua usando dei caratteri speciali. Ad esempio, se vogliamo salvare su file la lista delle variabili di shell e di ambiente, dobbiamo redirezionare l'output di set in questo modo:

```
[pippo@localhost pippo]$ set > lista.txt [-> Invio]
```

La lista delle variabili non viene più visualizzata, ma nella directory è stato creato un file lista.txt che contiene proprio quello che volevamo. Il carattere `>` indica a set che non deve scrivere su sdtout, ma su `lista.txt`; `>` costringe set a sovrascrivere il file `lista.txt`, ma è possibile anche far sìche i caratteri vadano ad accodarsi alla fine del file `lista.txt` usando `>>`. Se invece vogliamo redirezionare lo standard error, dobbiamo utilizzare `2> nomefile`, dove il 2 sta

ad indicare il secondo canale di output che è proprio quello di stderr. Se vogliamo redirezionare entrambi i flussi di output ad un unico file, possiamo utlizzare `&>`. Per quanto riguarda lo standard input, il carattere speciale da utilizzare per la redirezione è `<`. Ad esempio, è possibile ordinare il contenuto di un file col comando sort come segue:

```
[pippo@localhost pippo]$ sort < elenco.txt [-> Invio]
abaco
babbione
zuzzurellone
```

In DOS, per generare la lista ordinata dei file di una directory esiste l'opzione /O del comando DIR. Il suo analogo Linux è il comando ls, come abbiamo già visto, che però non possiede questa opzione. Si potrebbe ovviare al problema usando il comando sort in questo modo:

```
[pippo@localhost pippo]$ ls -l > elenco.txt [-> Invio]

[pippo@localhost pippo]$ sort < elenco.txt [-> Invio]
```

il primo comando crea un file di nome `elenco.txt` che contiene la lista dei file disordinata, il secondo comando visualizza sullo schermo la lista ordinata. È una soluzione un po' troppo scomoda per risolvere un problema così frequente, e infatti Bash mette a disposizione una scorciatoia per effettuare questa operazione: la pipe.

In inglese pipe significa "tubatura, condotto"; questo concetto è proprio quello che serve per risolvere il probema di prima. Infatti noi vogliamo che l'output disordinato di ls venga letto da sort per essere poi ordinato. Ovvero, vorremmo fare una "tubatura" che porta il flusso di caratteri (non d'acqua...) da ls in sort. La soluzione manuale già vista, cioé quella di creare un file intermedio (`elenco.txt`) è un modo per creare questo condotto, ma le pipe eseguono questa operazione automaticamente. In pratica, si scrive:

```
[pippo@localhost pippo]$ ls | sort [-> Invio]
```

dove il carattere `|` rappresenta proprio un tubo stilizzato. Questo comando effettua tutte le operazioni che prima facevamo a mano, ovvero prende l'output di ls e lo scrive su un file di tipo speciale (detto tipo pipe, appunto) gestito con una politica a coda, cioè il primo dato scritto sul file è il primo dato che viene letto poi da sort: come la coda alla posta! Una volta completata la creazione del file intermedio, sort legge tutti i dati e li ordina. Tutto questo accade senza che l'utente abbia alcun messaggio sullo schermo, perché le pipe sono silenziose. Finito l'ordinamento, finalmente la lista viene visualizzata sullo schermo.

Nel caso in cui la lista dei file sia troppo lunga per stare tutta in una schermata, sarebbe desiderabile poterla leggere una schermata alla volta. ls non prevede nemmeno questa possibilità, ma le pipe ci vengono ancora una volta in aiuto. Per dividere l'output di ls in pagine, di solito si utilizzano i comandi more o less: il primo aspetta la pressione di un tasto per visualizzare la pagina successiva, il secondo è più evoluto e permette anche di scorrere avanti e indietro di una riga o una pagina alla volta. Per ottenere una lista pagina dopo pagina, basta allora digitare:

```
[pippo@localhost pippo]$ ls | more [-> Invio]
```

oppure

```
[pippo@localhost pippo]$ ls | less [-> Invio]
```

1.8 Lista di alcuni comandi di frequente utilizzo

Help in linea (per visualizzare un testo esplicativo sui comandi)

`man nomecomando`
`info nomecomando`
`nomecomando --help`

Operazioni su directory

`cd nomedir`: (change directory) cambia la directory corrente
`mkdir nomedir`: (make directory) crea una directory
`rmdir nomedir`: (remove directory) cancella una directory solo se è vuota
`pwd`: (print working directory) visualizza il nome della directory corrente

Operazioni su file

`cp sorgente destinazione`: (copy) copia i file da `sorgente` a `destinazione`
`rm nomefile`: (remove) cancella i file
`mv sorgente destinazione`: (move) sposta i file da `sorgente` a `destinazione`
`chmod permessi nomefile`: (change mode) cambia i permessi di un file

Compressione e decompressione di file

`tar`: per lavorare con i file .tar
`gzip`: per lavorare con i file .gz
`zip`: per lavorare con i file .zip

Visualizzazione dei file (soprattutto nomi o contenuto)

`ls`: (list) visualizza la lista dei file di una directory
`find -name "nomefile"`: cerca un file nel filesystem
`less nomefile`: mostra il contenuto di un file una pagina alla volta
`more nomefile`: mostra il contenuto di un file una pagina alla volta
`sort`: ordina i dati che riceve

Informazioni sui dischi

`df`: (disk free) mostra la quantità di spazio libero nelle varie partizioni montate
`du nomedir`: (disk used) mostra la quantità di spazio occupato da una directory
`mount filesystem nomedir`: monta il filesystem nella directory specificata

Gestione degli utenti

`adduser nomeutente`: aggiunge un utente
`passwd nomeutente`: cambia la password di un utente
`su nomeutente`: (substitute user) cambia l'utente attivo

Comandi particolari

`echo messaggio`: visualizza un messaggio

`alias nuovonomecomando="comando"`: dà un altro nome al comando specificato
`unalias nomealias`: elimina un alias
`set nomevariabile="valore"`: imposta il valore per la variabile specificata
`unset nomevariabile`: elimina una variabile
`export variabile`: trasforma una variabile di shell in una di ambiente

1.9 Approfondimenti sulla shell

Per maggiori informazioni sui comandi, esiste in Bash un ottimo help in inglese che funziona come il comando HELP di DOS: è `man`. Per utilizzarlo digitare

```
[pippo@localhost pippo]$ man nomecomando [-> Invio]
```

È scritto in lingua inglese, ma è la fonte principale di informazioni per chi vuol conoscere a fondo i comandi della shell.

Un ulteriore lettura caldamente consigliata è l'HOWTO DOS2Linux, scritto sia in italiano che in inglese, reperibile al sito www.pluto.linux.it alla voce documentazione.

1.10 Bibliografia di uso generale

Sono molte le fonti da cui trarre informazioni su Linux, e in questa sede riporterò le più famose.

1.10.1 Help forniti con Linux

da console ci sono "man" e "info" dei quali si è visto l'utilizzo, in KDE ci sono la Guida di KDE e I Suggerimenti di Kandalf

1.10.2 Internet

- www.pluto.linux.it (alla voce Documentazione, si possono trovare gli HOWTO in italiano e Appunti di Informatica Libera)
- www.linux.com (per tutte le utenze, fornisce anche gli HOWTO)
- www.linuxplanet.com
- www.gnu.org (per avere informazioni su cos'è il software libero)
- www.linuxnewbie.com (per principianti)
- www.mandrakesoft.com (è il sito della distribuzione Mandrake, la più adatta a chi si avvicina al mondo Linux. Vedere al link Documentazione)

Capitolo 2

I Processi

di Carlo Pinciroli

2.1 Cenni generali sui processi

In Unix, il termine processo si utilizza per indicare un programma in esecuzione in memoria e il suo ambiente associato, ovvero l'insieme dei file di input e output che gli appartengono e delle sue variabili di ambiente.

Un processo è quindi un programma avviato nella sessione di lavoro: perciò anche la shell Bash è un processo. Nuovi processi possono essere creati solo in risposta alla richiesta di un processo già esistente. Ad esempio, quando scriviamo un comando da terminale, la shell Bash crea un nuovo processo utilizzando il comando `fork()` messo a disposizione dal kernel (i comandi di questo genere sono detti in gergo "chiamate di sistema"). Il comando `fork()` (in inglese "biforcazione") permette a un processo di creare una copia di se stesso, che viene poi utilmente modificata per fare spazio al nuovo processo che si vuole avviare. Ad esempio, quando mandiamo in esecuzione un comando dalla shell, quest'ultima crea una copia di se stessa (in modalità non interattiva) e poi esegue il comando digitato.

Il meccanismo della biforcazione induce quindi fra i processi un'organizzazione di tipo padre-figlio che genera una struttura ad albero: ci si può rendere conto di questo fatto digitando pstree al prompt dei comandi. pstree visualizza proprio l'albero dei processi con lo stesso aspetto con cui di solito si visualizza la struttura delle directory di un filesystem. Cosìcome l'albero delle directory possiede una radice, anche l'albero dei processi ne deve avere una: la radice è infatti il processo init, che viene attivato direttamente dal kernel in fase di avvio; init poi crea tutti gli altri processi del sistema con le fork() necessarie.

Quando un processo figlio termina la propria esecuzione, lo fa tramite la chiamata di sistema exit(); il processo figlio diventa allora uno "zombie", perché sebbene sia terminato non è ancora stato rimosso dalla memoria. E' il processo padre che si occupa di rimuoverlo dalla memoria, facendone scomparire ogni

traccia. Se il processo terminato aveva dei figli non ancora terminati, questi vengono affidati al processo init.

Evidentemente, durante una sessione di lavoro sono molti i processi attivi che lavorano in contemporanea. Basti pensare a quanti processi sono legati all'utilizzo dell'ambiente grafico (quelli del server grafico e dei vari client in esecuzione) oppure ai vari comandi che si possono combinare insieme in un'unica riga facendo uso delle pipe. Inoltre molti altri programmi "sotterranei" sono attivi anche se non si notano tracce delle loro azioni. Tutte queste attività contemporanee devono essere svolte dall'unica (di solito) CPU di cui le macchine sono provviste: tanto lavoro per un solo operaio! Per questa ragione, il kernel fa sìche un processo venga eseguito per un po' di tempo e che, terminato tale tempo, il processo venga messo in attesa per lasciare spazio ad un altro processo, che viene eseguito anche lui per una certa quantità di tempo: quindi la CPU continua a eseguire un po' alla volta i vari processi, passando da uno all'altro senza sosta. Il tempo che la CPU dedica ad ogni processo non è sempre lo stesso, ma dipende dalla priorità (ovvero dall'importanza) che il processo attivo possiede. La priorità è immagazzinata in un numero, che funziona in modo opposto a come ci si aspetterebbe: infatti un basso valore indica un'alta priorità, mentre un valore alto indica una priorità bassa. Il valore di priorità può essere modificato facendo uso di un altro numero, detto valore di nice, che può essere positivo o negativo. Il nice viene sommato alla priorità, quindi se nice è positivo il processo diventa meno importante e quindi ha meno tempo a disposizione, ragion per cui risulta rallentato.

Un processo ha diversi stati mutuamente esclusivi in cui può trovarsi: in esecuzione, in attesa di un evento (ad esempio la pressione di un tasto o la ricezione di dati da disco), sospeso, o zombie.

Infine, ogni processo ha generalmente i permessi dell'utente che l'ha avviato.[1] Questo fatto è una delle ragioni che sta alla base della sicurezza di Linux. Se per ipotesi infatti il sistema venisse infettato da un virus che cancella i file di avvio, tale virus avrebbe i permessi dell'utente che ne ha subito l' infezione. Se l'utente infettato fosse root, allora il virus avrebbe diritto di fare qualunque cosa, ma se l'utente non fosse root, il virus non potrebbe fare nulla. Ecco, ancora una volta, una ragione per lavorare come utente normale e non come root.

2.2 Situazione dei processi: /proc e comandi associati

Come è stato già accennato, ad ogni processo sono associate delle informazioni utili per il kernel al fine di gestire l'organizzazione del lavoro del sistema. Le informazioni sono contenute in memoria e il kernel le mette a disposizione dell'utente perché possa consultarle.

Come è prevedibile, le informazioni sono immagazzinate in file fittizi ("Tutto è un file"...) distribuiti nella directory `/proc` che è un vero e proprio filesystem contenuto in memoria: non occupa quindi spazio fisico su disco, nonostante il kernel permetta di lavorarci come se fossero file e directory qualunque.

[1] Questo comportamento può essere modificato mediante l'utilizzo del bit SUID o del programma sudo, che permettono di far partire un processo con i permessi di un altro utente.

Il filesystem `/proc` è molto importante perché, oltre alle informazioni sui processi, fornisce anche notizie di altro genere sul sistema, come il tipo di hardware installato e la sua configurazione. Saper leggere i file che contengono questi dati è fondamentale per risolvere molti problemi di conflitti hardware, o solo per conoscere che tipo di componenti sono stati riconosciuti con successo dopo l'installazione del sistema operativo.

Ogni processo possiede una sottodirectory di `/proc` il cui nome è il PID. All'interno della directory relativa ad un processo, ci sono dei file, ognuno dei quali contiene varie informazioni. Non è necessario leggere il contenuto di questi file, perché è un compito effettuato da alcuni comandi messi a disposizione dalla shell, che hanno il pregio di essere più comodi da usare. Inoltre, l'evoluzione continua dei kernel fa sìche nelle nuove versioni il modo di gestire il filesystem `/proc` sia diverso, e così l'utilizzo di comandi che siano in grado di coprire queste differenze dal punto di vista dell'utente sono fondamentali.

2.2.1 Visualizzare la situazione dei processi

Per ottenere informazioni sulla situazione dei processi, sono disponibili tre comandi:

- `ps`: questo comando visualizza la "fotografia" della situazione dei processi al momento in cui si è digitato ps e premuto invio. Se viene digitato senza argomenti, visualizza solo la lista dei processi avviati dall'utente:

```
[pippo@localhost pippo]$ ps           [-> Invio]
  PID TTY          TIME CMD
  1651 pts/1    00:00:00 bash
  1760 pts/1    00:00:00 ps
```

  La colonna PID è già stata spiegata; TTY è il terminale da cui si è avviato il processo (nell'esempio il terminale 1); TIME è il tempo di utilizzo della CPU; CMD è il nome dell'eseguibile associato al processo. L'opzione a (ovvero "all", tutti; con o senza trattino è lo stesso) permette di vedere tutti i processi di tutti gli utenti:

```
[pippo@localhost pippo]$ ps a          [-> Invio]
(non mostro l'output perche' e' molto lungo)
```

  Invece l'opzione l ("long", lungo) permette di vedere un elenco più dettagliato:

```
[pippo@localhost pippo]$ ps l          [-> Invio]
  F   UID   PID  PPID PRI  NI    VSZ  RSS WCHAN  STAT TTY        TIME COMMAND
000   501  1651  1650  11   0   2900 1724 wait4  S    pts/1      0:00 /bin/bash
000   501  1761  1651  19   0   3172 1224 -      R    pts/1      0:00 ps l
```

  dove UID è lo User IDentificator (il numero che caratterizza un utente); PPID è Parent PID, ovvero il PID del processo genitore; PRI è la priorità del processo; NI è il valore nice, ovvero il valore che sommato a PRI permette di modificare la priorità del processo; RSS indica l'occupazione

effettiva della memoria; WCHAN è l'evento di cui è in attesa il processo; STAT è lo stato del processo; COMMAND è la riga di comando che ha avviato quel processo.

- `pstree`: questo comando è già stato illustrato: visualizza la lista dei processi organizzata ad albero, di modo che la parentela fra i vari processi risulti visivamente evidente.

```
[pippo@localhost pippo]$ pstree          [-> Invio]

init-+-artsd
     |-atd
     |-crond
     |-devfsd
     |-gpm
     |-kapm-idled
     |-7*[kdeinit]
     |-kdeinit-+-3*[kdeinit]
     |         `-kdeinit---bash---pstree
     |-kdeinit---cat
     |-keventd
     |-khubd
     |-klogd
     |-login---bash---startx---xinit-+-X
     |                               `-startkde---ksmserver
     |-mdrecoveryd
     |-5*[mingetty]
     |-syslogd
     `-xfs
```

- `top`: è come un ps aggiornato continuamente. Viene mostrata una schermata simile a quella che mostra il comando ps l, solo che top continua ad aggiornarla. È possibile anche inserire dei comandi interattivi costituiti da un carattere singolo. h serve per richiamare l'help sui comandi.

2.3 Comunicazione fra processi: i segnali

Può capitare che, durante un lavoro, ci sia un processo in esecuzione che rallenta l'elaborazione, o che un processo bloccato influenzi negativamente la velocità del sistema. Quando questo accade, è desiderabile poter intervenire sul processo che genera il malfunzionamento e avere la possibilità di metterlo in pausa o anche di costringerlo a terminare. In Windows, per eseguire questa operazione si fa uso del Task Manager, ma spesso quando si impartisce il comando "Termina applicazione" vengono visualizzate una serie interminabile schermate blu che informano che il programma non risponde o che addirittura il sistema è diventato instabile. Questo accade perchè in Windows la terminazione forzata di un programma porta con sè degli strascichi legati al fatto che il multitasking non è genuino. Invece Linux è un sistema veramente multitasking, e quindi è possibile terminare un processo senza compromettere la stabilità di tutto il sistema né i processi che non hanno legame col processo terminato.

Per costringere un processo attivo a mettersi in pausa o a terminare, è necessario inviargli un segnale, ovvero un messaggio che il processo è in grado di ricevere ed intepretare. L'utilizzo dei segnali è molto vasto e va ben oltre l'esempio qui riportato perché la comunicazione fra i processi è alla base della gestione dei processi in Linux.

I segnali disponibili sono moltissimi, ma i più importanti sono due: SIGKILL e SIGTERM. Ce ne sono altri tre che vedremo nel prossimo paragrafo. Quando un processo riceve il segnale SIGKILL, termina immediatamente la propria esecuzione senza salvare nulla del proprio lavoro fino a quel punto. Anche SIGTERM costringe un processo a terminare, ma in modo meno improvviso del precedente perché permette di salvare i dati prima della terminazione.

Per inviare messaggi ai processi, bisogna naturalmente fare riferimento al loro PID. Il comando kill permette di mandare i segnali in questo modo:

```
[pippo@localhost pippo]$ kill -s SIGKILL 1651          [-> Invio]
```

questo comando invia il segnale SIGKILL al processo il cui PID è 1651. Però è anche possibile digitare:

```
[pippo@localhost pippo]$ kill -s 9 1651          [-> Invio]
```

questo comando invia il segnale numero 9 al processo il cui PID è 1651. Il segnale numero 9 è SIGKILL. Per conoscere i numeri relativi ai vari segnali, si può utilizzare ancora kill:

```
[pippo@localhost pippo]$ kill -l          [-> Invio]
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR      31) SIGSYS      32) SIGRTMIN    33) SIGRTMIN+1
34) SIGRTMIN+2  35) SIGRTMIN+3  36) SIGRTMIN+4  37) SIGRTMIN+5
38) SIGRTMIN+6  39) SIGRTMIN+7  40) SIGRTMIN+8  41) SIGRTMIN+9
42) SIGRTMIN+10 43) SIGRTMIN+11 44) SIGRTMIN+12 45) SIGRTMIN+13
46) SIGRTMIN+14 47) SIGRTMIN+15 48) SIGRTMAX-15 49) SIGRTMAX-14
50) SIGRTMAX-13 51) SIGRTMAX-12 52) SIGRTMAX-11 53) SIGRTMAX-10
54) SIGRTMAX-9  55) SIGRTMAX-8  56) SIGRTMAX-7  57) SIGRTMAX-6
58) SIGRTMAX-5  59) SIGRTMAX-4  60) SIGRTMAX-3  61) SIGRTMAX-2
62) SIGRTMAX-1  63) SIGRTMAX
```

l'opzione -l (list) fa sì che vengano visualizzati tutti i possibili segnali preceduti dal loro numero identificativo.

Per inviare un segnale a tutti i processi che eseguono uno stesso comando si può fare uso di killall:

```
[pippo@localhost pippo]$ killall -KILL ls          [-> Invio]
```

Invia il segnale SIGKILL a tutti i processi creati col comando ls.

2.4 Processi e shell

Attraverso la shell, come abbiamo già detto, è possibile avviare dei processi, anche più di uno alla volta. Un comando contente delle pipe, ad esempio, o uno script di shell, generano più processi legati fra loro contemporaneamente. Questi tipi di insiemi di processi vengono chiamati job (ovvero "lavori") di shell. Non bisogna dunque confondere un processo con un job di shell: un processo è un singolo eseguibile avviato, un job è l'insieme dei processi avviati con un comando della shell.

I job possono operare in primo piano (foreground) o in secondo piano (background). La differenza fra le due modalità sta nel fatto che nel primo caso il terminale non è disponibile ad accettare nuovi comandi, nel secondo caso invece sì.

Evidentemente, mettere in background un job che richiede una frequente interazione con l'utente non è molto utile in quanto il fine di mettere in background un job è proprio la liberazione del terminale. È altrettanto chiaro che un job in background che produce molti messaggi sullo schermo non è consigliabile, in quanto non si potrebbero distinguere i suoi messaggi dai messaggi di un altro job in secondo piano o in primo piano che opera contemporaneamente. È quindi importante redirigere l'output dei job in background.

2.4.1 Avvio di un job sullo sfondo

Per avviare un job esplicitamente in background, è sufficiente far seguire al comando il carattere &:

```
[pippo@localhost pippo]$ find / -name "pippo" &    [-> Invio]
[1] 1856
```

I due numeri visualizzati indicano il numero del job e il numero dell'ultimo processo tra quelli attivati col comando. Se l'output non è stato rediretto, quando il job deve mostrare sullo schermo qualcosa viene visualizzato il messaggio

```
[1]+ Stopped (tty output) find
```

che indica che il processo è in attesa di poter scrivere sullo schermo. Se invece non è stato rediretto l'input, il messaggio

```
[1]+ Stopped (tty input) find
```

indica che il job aspetta di ricevere dalla tastiera dell'input.

2.4.2 Invio di segnali ad un job in primo piano

Oltre al comando kill già trattato, è possibile inviare dei messaggi ai job in foreground facendo uso di particolari combinazioni di tasti.

Premendo Ctrl+C si invia un segnale SIGINT che interrompe il processo attivo in primo piano, senza possibilità di recuperarlo. Se invece si preme Ctrl+Z si ottiene la sospensione del job (segnale SIGSTP), che rimane così congelato fino a che un comando fg o bg non gli invii un segnale SIGCONT. Nel caso in cui si sia digitato fg, il job riprende l'esecuzione in foreground; se invece è stato digitato bg, il job riprende l'esecuzione in background.

Il comando jobs mostra la lista dei job attivi sullo sfondo o sospesi. Esiste anche un doppione del comando kill, che permette di inviare ai job dei segnali come si faceva con i processi, anche se la sintassi è leggermente diversa:

```
[pippo@localhost pippo]$ kill -9 %1          [-> Invio]
```

invia il segnale 9 (SIGKILL) al job numero 1.

Capitolo 3

Dispensa sugli script

di Carlo Pinciroli

3.1 Informazioni preliminari sulla programmazione

L'utilizzo del computer è subordinato ai programmi di cui si dispone: per effettuare una particolare attività, è necessario che l'utente interagisca con un programma appositamente realizzato. La programmazione è proprio l'atto di creare un programma che soddisfa le richieste di chi lo deve utilizzare. I problemi fondamentali, nell'atto del programmare, sono dati da due fattori:

1. che la macchina non "pensa" come un umano, e quindi è necessaria una forma di astrazione da parte del programmatore al fine di avvicinare il proprio modo di ragionare a quello della macchina

2. una volta assunta questa disposizione, è necessario saper progettare, prima ancora che realizzare, una soluzione al proprio problema che sia attuabile da un computer.

In altre parole, le difficoltà della programmazione stanno nell'imparare come scrivere programmi che il computer può eseguire, e nel saperli progettare correttamente, cioè il più possibile privi di errori (bug). Solitamente, i neofiti della programmazione pensano che il primo dei due problemi sia il più importante. In realtà, una volta imparato un linguaggio di programmazione (che sia il C, il Java, il Visual Basic, il Pascal...) le difficoltà maggiori affiorano nella fase di progettazione.

Un linguaggio di programmazione è una vera e propria lingua che viene utilizzata per descrivere al computer le operazioni da eseguire. Il primo problema a cui si faceva riferimento poc'anzi, era che il computer usa un linguaggio diverso da quello umano. Comandi come "spegni il computer" o "stampa il documento" non hanno alcun significato per la macchina. Il computer utilizza il cosiddetto linguaggio macchina, che è un insieme di comandi formati dai soli caratteri 0 e 1.

Perché proprio 0 e 1 ? Perché le parti elettroniche del computer, per la loro stessa costruzione, sono impostate in modo da reagire a due soli livelli di tensione elettrica: uno basso (lo 0) e uno alto (l'1). Così, comandi come 0010011001010 (detti binari) sono interpretabili dal computer perché sono eseguibili dalle sue componenti elettroniche. Naturalmente programmare in questa maniera rende praticamente impossibile realizzare anche il più semplice programma, in quanto basterebbe confondere un 1 con uno 0 per rovinare tutto il proprio lavoro. Inoltre, programmare in linguaggio macchina (cioè impartire comandi direttamente ai componenti elettronici) richiede la conoscenza di alcuni particolari costruttivi, cosa anche questa del tutto fuori portata per un qualsiasi progetto, dal momento che il computer è pieno di componenti elettronici e le marche sono molte e tutte con differenti particolari di funzionamento. Si è deciso allora di realizzare dei linguaggi più vicini alla logica del programmatore umano, cosicchè le difficoltà descritte non sussistano più. Il C, il Pascal, il Java sono solo alcuni degli esempi di queste categorie di linguaggi, che vengono detti ad alto livello perché più vicini alla logica umana del linguaggio macchina e dell'assembler (detti a basso livello). Nei linguaggi ad alto livello le espressioni sono perciò ben diverse dagli incomprensibili 01001010 di cui parlavamo prima. Un esempio molto semplice è il famoso programma che visualizza sullo schermo il messaggio "Hello, world!", che scritto in C risulta:

```
void main()
{
  printf("Hello, world!");
}
```

Anche se non tutto quello che è scritto è chiaro a prima vista, si intuisce senza alcuno sforzo che l'operazione di visualizzazione del messaggio è effettuata dal comando `printf("Hello, world!")`: una parola che è ben più facilmente interpretabile di una serie di 0 e di 1...

Sorge dunque un grosso problema. Se il computer capisce solo 0 e 1, ma noi gli impartiamo comandi formati da parole, come è possibile far sìche questi comandi possano essere capiti dalla macchina? Ci sarebbe bisogno di una sorta di traduzione dal livello alto dell'utente a quello basso della macchina. Si ripresenta sotto altra forma, allora, la difficoltà sul modo di "pensare" del computer. I modi per realizzare questa traduzione sono fondamentalmente due, e li spiegheremo con un semplice esempio. Supponiamo di essere il presidente di una piccola ditta che produce componenti in metallo, e che una multinazionale giapponese sia interessata all'acquisto di tali componenti. Il presidente non conosce il giapponese, e dunque non è in grado di parlare direttamente con il delegato giapponese. Evidentemente c'è bisogno di una traduzione per potersi accordare. Si può allora procedere in due modi. Il primo modo è di redigere un testo in italiano, incaricare qualcuno di tradurlo e poi farlo leggere al delegato. Il secondo è di parlare direttamente col delegato servendosi di una traduzione simultanea: ogni frase pronunciata viene immediatamente tradotta dall'interprete.

Al di fuori dell'esempio, dunque, i modi per effettuare la traduzione da linguaggio ad alto livello a linguaggio a basso livello sono due. Si può prendere l'intero programma originale ed effettuarne una traduzione completa in binario: in questo modo si ottiene un programma perfettamente funzionante, il tipico `.EXE` del dos. Altrimenti si può leggere una riga di programma, tradurla, eseguirla, e poi leggere la riga successiva, tradurla, eseguirla e cosìvia fino alla fine

del programma. Nel primo caso si parla di linguaggi compilati, nel secondo caso di linguaggi interpretati. Sono compilati il C e il Pascal, ad esempio. Un famoso linguaggio intepretato è invece il qbasic che veniva fornito con il dos 6.22 e precedenti. Il Java invece, per soddisfare le esigenze di portabilità che la rete Internet richiede, è per metà compilato e per metà interpretato.

3.2 Gli script della shell

Gli script della shell sono delle specie di programmi, in quanto la logica con la quale si realizzano è la stessa dei programmi, ma non offrono le tutte le funzionalità di un vero e proprio linguaggio di programmazione. Gli script, nello svolgimento di alcune ripetitive attività, hanno un ruolo di spicco, in quanto permettono all'utente di automatizzare delle operazioni in modo banale. Nella loro forma più semplice, uno script non è altro che un file di testo in cui ogni riga è un comando riconoscibile da Bash: quando lo script viene eseguito, Bash esegue i comandi uno alla volta. Gli script sono quindi interpretati e non compilati. Vedremo i vari aspetti degli script facendo uso di alcuni esempi.

3.2.1 Il primo script

Potrebbe capitare di avere il disco pieno, e di voler quindi cancellare dei grossi file inutilizzati da tempo. Un comando che crea un elenco di questi file è il seguente:

```
[pippo@localhost pippo]$ find / -type f -atime +30 -size \
+1024k -exec ls -l {} \; > /tmp/largefiles    [-> Invio]
```

questo comando crea il file largefiles che contiene tutti i file più grandi di 1 megabyte e inutilizzati da almeno 30 giorni. Dover riscrivere periodicamente un comando del genere è piuttosto noioso, e se magari non se neè nemmeno ben capita la sintassi ricordarlo è del tutto impossibile. Una buona soluzione potrebbe essere la creazione di uno script che contiene proprio quel comando:

```
--- inizio dello script ---
#!/bin/bash
# uno script per creare un elenco di file voluminosi e
# inutilizzati da almeno un mese: vedere il file largefiles

find / -type f -atime +30 -size +1024k -exec ls -l {} \; > \
/tmp/largefiles
--- fine dello script ---
```

La prima riga contiene un'informazione basilare per la shell: infatti dice qualedeve essere l'interprete dei comandi, nel nostro caso naturalmente Bash. Ma nel caso si faccia uno script in qualche altro linguaggio, ad esempio Perl, quella riga dovrà contenere il percorso dell'eseguibile perl. La seconda riga è un commento, in quanto inizia con un #, ed è utile per descrivere cosa fa lo script: quando lo rileggeremo, se non ci ricorderemo cosa significano i comandi, almeno sapremo cosa aspettarci dallo script. La terza riga è il comando che volevamo e non richiede ulteriori spiegazioni. Possiamo utilizzare un editor qualunque (vi, emacs, o altro ancora) per creare un file di testo che contenga quelle righe e

salvarlo col nome filegrandi. A questo punto, una volta creato il file contentente lo script, non possiamo ancora eseguirlo. Infatti, se controlliamo col comando ls -l filegrandi, vedremo:

```
[pippo@localhost pippo]$ ls -l filegrandi          [-> Invio]
-rw-rw-r--    1 pippo    pippo       203 feb  8 20:05 filegrandi
```

per qualunque utente, non esiste il diritto di esecuzione! Quindi dobbiamo prima impostare questo diritto, col comando chmod:

```
[pippo@localhost pippo]$ chmod +x filegrandi          [-> Invio]
[pippo@localhost pippo]$ ls -l filegrandi          [-> Invio]
-rwxrwxr-x    1 pippo    pippo      203 feb  8 20:05 filegrandi
```

C'è ancora un problema: per eseguire lo script non si può scrivere

```
[pippo@localhost pippo]$ filegrandi          [-> Invio]
bash: filegrandi: command not found
```

perchè la directory corrente non c'è fra le directory del path: il comando giusto è

```
[pippo@localhost pippo]$ ./filegrandi          [-> Invio]
```

in cui il ./ iniziale specifica che filegrandi si trova nella directory corrente. Abbiamo cosìrealizzato il nostro primo script, e potremo riutilizzarlo tutte le volte che vorremo. Potrebbe essere una buona idea creare una directory personale `/scripts` in cui mettere le nostre creazioni e aggiungerla al path, cosicché il problema del `./` iniziale non si ripresenterebbe più.

3.2.2 Parametri

I file di testo in DOS hanno come marcatore di fine riga una sequenza di due caratteri: CR e LF (carriage return e line feed, ovvero ritorno carrello e nuova linea). In Linux invece si usa soltanto carattere LF, quindi se si visualizza un file di testo in formato DOS usando less o more si noteranno dei caratteri M̂ a fine riga: sono proprio i CR che Linux non usa per andare a capo, e perciò li tratta come se fossero caratteri qualunque. Evidentemente sarebbe bello poter eliminare questi caratteri, e Linux mette a disposizione il comando tr per farlo. Se il file `testodos.txt` è in formato dos e si vuole creare un file di testo senza i CR di nome `testolinux.txt`, si può scrivere:

```
[pippo@localhost pippo]$ tr-d '\015' < testodos.txt > \
testolinux.txt          [-> Invio]
```

Anche in questo caso, il comando in questione richiede di ricordare dei particolari non proprio intuitivi: il fatto che si debba usare il comando tr, raramente usato nel lavoro di routine, e che il carattere CR abbia il codice ASCII 015. Ci sono tutte le condizioni per realizzare un comodo script dal nome semplice, ad esempio dos2unix. L'utilizzo che ne vorremmo fare è il seguente:

```
[pippo@localhost pippo]$ ./dos2unix testodos.txt \
testolinux.txt          [-> Invio]
```

cioè vorremmo che il nostro script accettasse dei parametri: i nomi dei file sorgente e destinazione. Bash permette di creare script che sappiano gestire i parametri, quindi il nostro script può essere realizzato:

```
--- inizio dello script ---
#!/bin/bash
# uno script per convertire i file di testo dos (con CR e LF)
# in file di testo linux (col solo LF)

tr-d '\015' < $1 > $2
--- fine dello script ---
```

nel comando tr, come si vede, al posto dei nomi dei file ci sono dei caratteri speciali: $1 e $2. Questi due indicano rispettivamente il primo argomento dello script, e il secondo argomento dello script. Quindi se si scrive sulla shell

```
[pippo@localhost pippo]$ ./dos2unix testodos.txt \
testolinux.txt          [-> Invio]
```

$1 vale `testodos.txt` e $2 vale `testolinux.txt`. Prima dell'esecuzione del comando tr nello script, Bash sostituisce $1 e $2 coi rispettivi valori e poi esegue il comando con le sostituzioni. Per gestire i parametri, dunque, basta utilizzare $N, dove N indica il numero del parametro. Se N è maggiore di 9, non si scrive ad esempio $10 ma ${10}. I parametri speciali invece sono i seguenti:

- $0 indica il nome dello script (nel caso precedente $0 valeva `dos2unix`)
- $* è la stringa formata dall'insieme di tutti i parametri (quindi nell'esempio precedente valeva `"testodos testolinux"`)
- $@ è l'insieme delle stringhe di tutti i parametri (cioè `"testodos" "testolinux"`)
- $# è impostata al numero di parametri specificati (nell'esempio 2)

Per provare tutto questo, si può creare uno script di nome vediamo_un_po:

```
--- inizio dello script ---
#!/bin/bash
# serve per provare i vari
# parametri speciali

echo "Il nome dello script e‘ $0"
echo "I parametri sono in tutto $#"
echo "Il primo parametro e‘: $1"
echo "Il secondo parametro e‘: $2"
echo 'Ecco il valore di $*:'
echo $*
--- fine dello script ---
```

3.2.3 Variabili

Abbiamo visto, nella lezione sulla Bash, l'utilità di poter disporre di variabili, che siano di shell o di ambiente. Questa utilità si ripresenta anche nella programmazione degli script. La dichiarazione e l'utilizzo delle variabili è lo stesso che abbiamo trattato precedentemente. Naturalmente, una variabile creata in uno script è valida soltanto all'interno dello script stesso (ovvero è una variabile di shell); per farla diventare una variabile di ambiente, bisogna utilizzare la consueta formula `"export variabile"`. All'interno degli script di boot (che tratteremo in futuro) ci sono molti esempi di dichiarazioni di variabili, uno per tutti la dichiarazione contenuta nel file /etc/rc.d/rc.sysinit:

```
--- inizio dello script ---
#!/bin/bash

# cose varie... le vedremo poi

# Set the path
PATH=/bin:/sbin:/usr/bin:/usr/sbin
export PATH

# cose varie... le vedremo poi
--- fine dello script ---
```

È anche possibile leggere dell'input da tastiera e immagazzinarlo in una variabile:

```
--- inizio dello script ---
#!/bin/bash
# legge da da tastiera e mostra
# quanto e‘ stato scritto

echo -n "Immettere una stringa:"
read STRINGA
echo "Hai inserito $STRINGA"
--- fine dello script ---
```

Il comando `echo -n` fa sì che dopo aver scritto `"Immettere una stringa:"` il cursore non vada a capo; il comando read legge la stringa e la immagazzina nella variabile `STRINGA`. Un fatto molto importante è che la variabile `STRINGA` non è stata dichiarata precedentemente: chi ha già programmato in QBasic non troverà questo fatto molto rilevante, ma i programmatori di C o simili potrebbero trovarlo sorprendente.

3.2.4 Costrutto if

Nello script `dos2unix` c'è la necessità di specificare i parametri. Infatti, se scriviamo soltanto

```
[pippo@localhost pippo]$ ./dos2unix          [-> Invio]
./dos2unix: $1: ambiguous redirect
```

il messaggio di errore che viene visualizzato non è molto chiaro per chi non sappia come è fatto lo script. Un buon programmatore deve però rendere utilizzabili a tutti le proprie creazioni: qui dovremmo visualizzare un messaggio di errore più chiaro, che spieghi il perché le cose non funzionano. L'effetto che vorremmo ottenere è di questo genere:

```
[pippo@localhost pippo]$ ./dos2unix          [-> Invio]
Utilizzo di questo script:
dos2unix filesorgente filedestinazione
```

cosìè sicuramente più chiaro! Quello di cui abbiamo bisogno all'atto della stesura dello script, quindi, è la possibilità di dire che se non ci sono parametri allora viene visualizzato il messaggio di errore, altrimenti il comando `tr` viene eseguito normalmente. Lo script che realizza questo effetto è il seguente:

```
--- inizio dello script ---
#!/bin/bash
# uno script per convertire i file di testo dos (con CR e LF)
# in file di testo linux (col solo LF)

if test $# = 2
then
  tr-d '\015' < $1 > $2
else
  echo "Utilizzo di questo script:"
  echo "$0 filesorgente filedestinazione"
fi
--- fine dello script ---
```

Il costrutto

```
if condizione
then
  lista_di_comandi
else
  lista_di_comandi
fi
```

realizza proprio quello che volevamo. Se la condizione dopo `if` è vera, viene eseguita la lista di comandi dopo `then`; altrimenti viene eseguita la lista di comandi dopo `else`. La parola `fi` marca la fine del costrutto. Nella condizione abbiamo messo il comando `test`. Questo è un comando che restituisce 0 se la condizione che lo segue `($# = 2)` è vera e restituisce un numero diverso da zero altrimenti. Quindi in realtà `if` è in grado di distinguere tra vero e falso utilizzando la distinzione tra 0 e non-0 ! In altri termini, per if lo 0 è vero e il non-0 è falso. Ogni comando, non solo test, restituisce un numero in uscita che dà informazioni sulla riuscita o meno dell'elaborazione. Ad esempio il comando `rm` restituisce 0 se riesce a fare il suo lavoro, cioè cancellare i file specificati, altrimenti restituisce un valore diverso da 0 che indica il tipo di errore che si è verificato. Nell'utilizzo normale della shell i valori restituiti dai comandi non sono molto importanti, e infatti non ne avevamo fatto menzione in passato, ma nello scripting l'utilità di questi valori è enorme. Vediamo infatti uno script che crea una copia di backup di un file di testo prima di avviare `vi` per modificarlo:

```
--- inizio dello script ---
#!/bin/bash
# uno script per creare una copia
# di backup del file da modificare con vi

if test $# = 0
then
  echo "Utilizzo di questo script:"
  echo "$0 nomefile"
else
  if cp $1 "~$1"
  then
    vi $1
  else
    echo "Copia di backup non riuscita"
  fi
fi
--- fine dello script ---
```

questo script è molto istruttivo. Ci sono due if, uno dentro l'altro (in gergo si dice "annidati"), e l'`if` interno è quello che controlla che `cp` non abbia avuto problemi nella creazione della copia di backup. Una particolare trattazione merita il comando `test`: serve, come abbiamo visto, per specificare delle condizioni unitamente al comando `if` (ma non solo, e lo vedremo poco più avanti). Tanto per cominciare, un consiglio da seguire alla lettera è di non chiamare mai un proprio script test. Questo perchè Bash potrebbe capire male, e confondere il vostro script `test` con il comando omonimo. Inoltre, `test` può essere utlizzato in modo molto più comodo di quanto non si sia visto finora. Infatti si può realizzare lo script `dos2unix`anche così:

```
--- inizio dello script ---
#!/bin/bash
# uno script per convertire i file di testo dos (con CR e LF)
# in file di testo linux (col solo LF)

if [ $# = 2 ]
then
  tr-d '\015' < $1 > $2
else
  echo "Utilizzo di questo script:"
  echo "$0 filesorgente filedestinazione"
fi
--- fine dello script ---
```

dove al posto di test abbiamo messo delle parentesi quadre. Questo modo di vedere il comando test è più usato e più comodo, quindi d'ora in poi lo utilizzeremo sempre. Alcune condizioni di utilizzo generale sono:

- `if [ -e nomefile ]` se esiste il file chiamato nomefile
- `if [ -f nomefile ]` se esiste ed è un file normale il file chiamato `nomefile`

- `if [ -d nomedir ]` se esiste ed è una directory il file chiamato `nomefile`
- `if [ -r nomefile ]` se esiste ed è leggibile il file chiamato `nomefile`
- `if [ -w nomefile ]` se esiste ed è modificabile il file chiamato `nomefile`
- `if [ -x nomefile ]` se esiste ed è eseguibile il file chiamato `nomefile`

per dire "se non..." si usa il carattere ! :

- `if ! [ -e nomefile ]` se non esiste il file chiamato `nomefile`
- `if ! mkdir pippo` se non è stato possibile creare la directory `pippo`

per sapere se una stringa di caratteri è uguale ad un'altra stringa (ad esempio se $1 vale `"-n"`):

```
if [ $1 = "-n" ]
```

per sapere se una stringa di caratteri è diversa da un'altra stringa (ad esempio se $1 non vale `"-n"`):

```
if [ $1 != "-n" ]
```

Per maggiori informazioni sul comando `test`, digitare `man test` al prompt dei comandi.

3.2.5 Costrutto while

Il costrutto `while` permette di eseguire un gruppo di operazioni in modo ripetitivo, finché non si verifica una condizione che fa terminare la ripetizione:

```
--- inizio dello script ---
#!/bin/bash

RISPOSTA="n"
while [ $RISPOSTA != "s" ]
do
  echo "Per farmi smettere devi scrivere s"
  echo -n "Immetti una stringa: "
  read RISPOSTA
  echo "Hai inserito $RISPOSTA"
done
--- fine dello script ---
```

3.2.6 Ciclo for

Anche il ciclo `for` permette di eseguire un gruppo di operazioni in modo ripetitivo finché non si verifica una condizione che fa terminare la ripetizione, ma il numero di ripetizioni in questo caso è fisso e non variabile come nel costrutto `while`. In altre parole, nel costrutto `while` è possibile far continuare il ciclo un numero indefinito di volte, mentre nel ciclo `for` il numero di volte è specificato in partenza.

```
--- inizio dello script ---
#!/bin/bash

for P in $*
do
  echo $P
done
--- fine dello script ---
```

questo script mostra uno per uno tutti gli argomenti digitati. Si può anche fare

```
for F in /home/pippo/*
do
  touch $F
done
```

in questo caso, per ogni file della directory `/etc` viene eseguito il comando `touch` che aggiorna la data di modifica del file alla data corrente.

3.3 Per avere maggiori informazioni

Nella stesura di questa dispensa, si è fatta l'ipotesi che il lettore non avesse alcuna esperienza di programmazione. Per questa ragione molti aspetti dello scripting sono stati omessi, in quanto sono o troppo complicati in un primo approccio o poco importanti al fine di capire lo scripting in generale. Alcuni degli argomenti omessi riguardano le variabili (modi di riferimento e array), varie opzioni di test, valutazione delle operazioni matematiche, costrutti case e select. Sono comunque argomenti affrontabili da soli senza difficoltà e non molto utilizzati nello scripting di routine. Nel caso in cui ci fosse l'esigenza di saperne di più, uno sguardo alle pagine di manuale può essere utile:

```
[pippo@localhost pippo]$ man bash          [-> Invio]
```

Altrimenti ci si può riferire alle fonti già elencate nella prima dispensa, con un particolare riguardo agli Appunti di Informatica Libera.

Capitolo 4

Boot del sistema

di Carlo Pinciroli Francesco Toso

4.1 Boot di sistema

Questa parte si occupa di descrivere ciò che accade a livello hardware nella macchina; non è necessaria per comprendere e modificare l'avvio di *Linux* o di un qualsiasi sistema operativo, ma può risultare utile conoscerla.

Osservazione: *Quanto segue cerca di essere il più generico possibile, tuttavia, in particolar modo le diagnostiche sulla corrente e simili vengono effettuate solo dall'introduzione dello standard* ATX *per l'alimentazione. Inoltre quanto scritto in questa sezione si intende valido solo per PC IBM o compatibili.*

Appena acceso il computer, dopo aver atteso che le correnti interne raggiungano il regime stazionario, viene avviato il *system BIOS* (o più brevemente *BIOS*) in modalità *superuser* del processore. Infatti, per convenzione all'avvio il processore ha impostato il registro *Instruction Pointer* all'indirizzo del *BIOS boot program*, che solitamente ha come indirizzo *FFFF0h*, proprio alla fine della memoria "base"[1] del sistema. Questo perché così è possibile mascherare la vera dimensione del *BIOS* che quindi può cambiare arbitrariamente da versione a versione.

Di seguito si elancano i passi principali dell'avvio con una breve spiegazione:

POST Il *POST*[2] notifica eventuali errori hardware, impedendo l'attivazione dei componenti.

Video Il *BIOS* si preoccupa, quindi di caricare ed eseguire il *BIOS* della sche-

[1] Tristemente questa è una eredità dei primi processori intel che non avevano ancora la modalità protetta del processore

[2] Power On Self Test, è una diagnostica veloce sul corretto comportamento elettrico dei componenti principali

da video[3] (solitamento posizionato all'indirizzo *C000h*) che contiene le istruzioni per l'inizializzazione della stessa.[4]

Altre ROM A questo punto vengono cercate le *ROM* degli altri dispositivi presenti sulla scheda come il *controlle IDE/ATA*, quello *SCSI* ma anche quello della scheda di rete, per esempio. Ciascuna di queste *ROM* viene poi eseguita secondo vari criteri di priorità.

Test Inizializzati tutti i device necessari, vengono effettuati test sul processore, sulla memoria e altre componenti critiche.

Inventory Vengono "ordinati" i device presenti e impostati i parametri degli stessi (ad esempio il *memory timing*, i parametri dei dischi...[5])

BOOT Finalmente, in base alle impostazioni date, viene determinato il dispositivo di *BOOT* e si cercano le informazioni per avviare il sistema operativo.[6]

Fail Nel caso di fallimento del *BOOT* viene mostrato un messaggio di errore

Quanto elencato è anche detto *cold boot* o avvio a freddo, il *warm boot* è identico a meno di quanto precede il *POST* (incluso) che viene saltato.

Una volta caricato il "core" del sistema operativo (ricordiamo che prima di tale caricamento il sistema non ha alcuna nozione di *file system*[7]), questo viene eseguito.

La sua esecuzione comporta l'inizializzazione dei device basilari[8] e la "messa in comunicazione" degli stessi col sistema operativo.

Infine viene caricato il *kernel* vero e proprio che, in generale, si occupa di interfacciarsi con le periferiche, di fornire i servizi dovuti e così via fino al caricamento dell'ambiente di lavoro dell'utente. Sotto *Linux*, caricato il kernel, il sistema cercherà di montare il *root filesystem*[9]; in caso di successo apparirà la seguente scritta:

```
VFS: Mounted root (ext2 filesystem) readonly.
```

Una volta fatto ciò, il kernel si preoccuperà di eseguire *init*.[10].

4.2 Boot Loader

Il *Boot loader* è il programma che si occupa all'avvio di indirizzare la ricerca del "core" del sistema operativo all'indirizzo opportuno del disco. In generale tale reindirizzamento è fisso per cui il *boot loader* è un semplice salto all'indirizzo

[3] Per questo motivo nei computer moderni prima si vede la diagnostica della scheda, poi il resto

[4] Un tempo, ma neanche tanto indietro, la gestione del video era demandata direttamente al processore per cui questa sezione risultava inesistente

[5] Si escludono le periferiche PnP che hanno una inizializzazione a parte

[6] Per i dischi fissi si usa l'*MBR* (*Master Boot Record*, C0, H0, S1), per i floppy il *Volume Boot Sector* (C0, H0, S1)

[7] Questo implica il fatto che risulti pressoché impossibile accedere ai dati non di boot del disco

[8] Ad esempio i due *bridge*, l'*AGP* port, i controller dei dischi...

[9] Si ricordi che sotto unix tutto è una directory...

[10] Si veda più avanti per informazioni più dettagliate a riguardo

ma nel caso di più sistemi operativi sulla macchina si richiede qualcosa di più complicato. Come potrete notare voi stessi, attualmente i *boot loader* hanno raggiunto livelli di complessità a volte eccessivi (vd. interfacce grafiche e simili), tuttavia il loro compito lo svolgono (e sono costretti a farlo) in maniera egregia.

Elenchiamo ora alcuni dei *boot loader* più comuni

- LILO
- Grub
- Boot magic
- NT loader

L'installazione di un *boot loader* in *MBR* piuttosto che su dischetto è solo questione di gusto, dato che ciò che concettualmente viene eseguito è sempre lo stesso.

Trattiamo brevemente i *boot loader* più famosi:

4.2.1 LILO

LILO è un *boot loader* multi uso. Non dipende da un particolare *file system*, può caricare le immagini del kernel *Linux* direttamente da floppy o disco fisso e può addirittura funzionare come *boot manager* per altri sistemi operativi come *DOS*, *Windows 9**, *SCO* .... Nella sua ultima versione è anche dotato di una gradevole interfaccia grafica.

4.2.2 Grub

Probabilmente il *boot loader* più giovane, ha dalla sua una estrema compattezza (solo qualche decina di KB) e una buona interfaccia grafica. Come *LILO*, può essere usato anche per altri sistemi operativi.

4.2.3 Boot magic

Il primo *loader* commerciale della lista, è abbastanza famoso perché fornito con *Partition Magic*. Ha dalla sua una interfaccia grafica accattivante e la possibilità di usare il mouse nelle scelte.

4.2.4 NT loader

Citato solo per dovere di cronaca, questo *boot loader* supporta solo sistemi operativi *Microsoft* per cui è completamente inutile per *Linux* e altri.

La scelta del *boot loader* è puramente arbitraria ed è solo una questione di gusti. A meno di esigenze particolari si consiglia di mantenere il *boot loader* della propria distribuzione.

Osservazione: *Si* ***sconsiglia vivamente*** *l'uso di* boot loader *in fase di test o poco conosciuti; i danni causabili da un programma del genere potrebbero essere anche gravi*

4.3 System V init

System V è la versione di *Unix* proposta dalla *AT&T* anni fa che ha dato origine ai sistemi *BSD*. Per motivi legali, infatti, la *AT&T*, quando ancora era solo una enorme compagnia telefonica, non potè commercializzare tale sistema che venne quindi donato liberamente all'università di Berkley.[11] Il *System V*, diventato ormai *386BSD*, venne quindi sviluppato sotto licenza *BSD*[12].
Dato che la comunità *Linux* aveva bisogno di costruire un modello per gli script di inizializzazione, venne preso come esempio quello del *System V* (che era diverso da quello di *BSD*).

Osservazione: *Si noti che distribuzioni come* RedHat *o* SuSE *utilizzino script* System V style*, mentre la* Slackware *usa script* BSD style*.*

4.3.1 Cenni generali

Una volta caricato in memoria, il kernel si preoccupa immediatamente di creare il processo init, che, come è già stato spiegato nella lezione sui processi, è il padre di tutti gli altri processi del sistema. "init" coordina il lavoro facendo uso delle informazioni contenute nel file */etc/inittab* che permette di definire gli script di avvio. Purtroppo ogni distribuzione ha un suo modo di organizzare il file /etc/inittab e gli script di avvio, quindi come esmpi porteremo quelli tratti dalla distribuzione RedHat (e quindi, anche sulla Mandrake che deriva da RedHat). In ogni caso le differenze tra distribuzione e distribuzione si concentrano soprattutto su particolari realizzativi (cambiano i nomi degli script o delle directory in cui sono posti, ad esempio), ragion per cui la logica che RedHat usa per avviare Linux non è per nulla differente da quella di Slackware o Debian.

4.3.2 Struttura di /etc/inittab

Dato che *inittab* è un file di configurazione di sistema, la convenzione delle directory di Linux impone che inittab si trovi nella directory /etc. "inittab", ovvero *INITialization TABle* (in inglese "tabella di inizializzazione") è il file in cui sono contenute le informazioni che *init* usa per impostare la configurazione del sistema e per avviare gli script di shell necessari. Ogni riga contiene un'impostazione, la cui sintassi generale è:

```
id:runlevel:action:process
```

Come si può notare ci sono vari campi separati dal simbolo ":", elenchiamo ora il significato di ciascuno

id: Identifica univocamente una riga del file di inizializzazione, puó essere costituito da un numero variabile di caratteri fino a un massimo di 2 o 4 a seconda delle librerie usate nella compilazione (*Linux* generalmente ne ammette un massimo di due)

runlevel: Indica il "livello di esecuzione" del processo, può contenere più caratteri per far eseguire il processo in più *runlevels*

[11] I veri motivi di tale azione sono tutt'ora oscuri ai più.
[12] Sostanzialmente la licenza permette enormi libertà sul codice.

action: Descrive che "azione" andrà intrapresa

process: È il processo da avviare

Runlevels

I "livelli di esecuzione" o runlevel sono una puro artificio software per permettere di avere una distinzione logica delle priorità e permessi dei processi in esecuzione sul sistema.

Osservazione: *Questo significa che il processore non necessariamente cambi priorità di esecuzione del processo in base al suo runlevel, nè che vada in modalità* superuser *a* runlevel *riservati (0,1 o 6).*

Il numero di *runlevel* è arbitrario, ma per convenzione in *Unix* è stato fissato a undici: da 0 a 9 più il livello S, ma solo i primi otto e il ivello S sono ufficialmente documentati in *Linux* e varianti di *Unix*.

Osservazione: *Si noti che il significato dei* runlevel *è puramente arbitrario e che ogni clone* unix *può implementarli come meglio crede. Addirittura tra diverse distribuzioni* Linux *spesso accade che ci siano differenze.*

Noi ci limiteremo a valutare i *runlevel* secondo la distribuzione *RedHat*, tuttavia è interessante notare che i *runlevel* 0, 1 e 6 sono riservati e hanno significati uguali per tutti mentre S non è accessibile direttamente:

runlevel 0 Questo è riservato all'arresto del sistema

runlevel 1 Corrisponde alla modalià monoutente

runlevel 6 È il livello di reboot

runlevel S È stato concepito per essere usato dagli script eseguiti in modalità monoutente

Interssante notare che in modalità monoutente è anche disponibile un terminale. Spesso si usa questa modalità quando ci sono problemi seri nel *boot* del sistema. Si sconsiglia vivamente, comunque, di usarlo, dato che tutti i processi vengono eseguiti coi privilegi di root, con le dovute conseguenze del caso.

Per quanto riguarda la distribuzione *RedHat* (e di riflesso *Mandrake*) sono di particolare interesse i livelli 3 e 5:

runlevel 3 Qui viene avviata la shell testuale ed è il runlevel a cui si opera solitamente

runlevel 5 A questo livello vengono eseguiti i processi del sistema X

I restanti sono usabili a piacimento.

Osservazione: *Passando da un* runlevel *ad un altro, tutti i processi in esecuzione che non sono indicati nel nuovo livello verranno eliminati prima tramite segnale* SIGTERM *poi con* SIGKILL *in caso di fallimento.*

Per ulteriori informazioni si consiglia di guardare "man init", "man inittab" e i readme sui runlevel sparsi nelle directory di init.

Action

Il campo *action* ammette numerosi valori, i più importanti sono:

respawn: Il processo verrà rieseguito non appena termina (ex. getty)

boot: Si eseguirà il processo al boot del sistema, il runlevel verrà ignorato

initdefault: Serve per impostare il livello di esecuzione predefinito non appena il sistema si avvia

wait: il processo viene avviato e *init* attende che termini prima di avviare il processo successivo

sysinit: il processo viene eseguito all'avvio del sistema prima dei "boot" vari

ctrlaltdel: il processo viene avviato quando init riceve il segnale SIGINT, che indica l'avvenuta pressione di Ctrl+Alt+Canc

Esempi

Alcuni esempi presi dal vero file inittab chiariranno la sua struttura, ad esempio

```
id:3:initdefault:
```

imposta il livello di esecuzione a 3, ovvero all'avvio viene eseguito *getty* in modalità testo, non grafica. Per poter usare X bisogna allora eseguirlo digitando startx al prompt dei comandi. Se il livello fosse stato 5, avremmo avuto una autenticazione grafica col conseguente avvio automatico di X.

```
si::sysinit:/etc/rc.d/rc.sysinit
```

Questa riga viene eseguita esattamente all'avvio del sistema come specificato da *sysinit*; il livello di esecuzione viene ignorato. Lo script "rc.sysinit", specificato ne campo *process*, è eseguito e serve per impostare molte variabili di ambiente (come HOSTNAME, ad esempio) ed eseguire degli script di configurazione.

Le seguenti righe indicano di eseguire lo script "rc" rispettivamente con parametro 0 o 3 a seconda del *runlevel* in cui si è entrati

```
l0:0:wait:/etc/rc.d/rc 0
l3:3:wait:/etc/rc.d/rc 3
```

Questo, invece, indica come comportarsi quando si rileva la pressione contemporanea dei tasi "ctrl, alt, del".

```
ca::ctrlaltdel:/sbin/shutdown -t3 -r now
```

Nulla ci vieterebbe di far apparire la scritta "Ha, ha!!Non puoi resettare!!" semplicemente chiedendo di fare echo sul terminale corrente alla pressione dei tre tasti, ma *cui prodest*?

Diamo un breve sguardo ora, ai terminali.

```
\# Run gettys in standard runlevels
1:12345:respawn:/sbin/mingetty tty1
2:2345:respawn:/sbin/mingetty tty2
...
```

Questo significa che i terminali 1,2...devono esistere ai *runlevel* 2,3,4 e 5 (il primo esiste anche a rl 1, altrimenti non sarebbe possibile la modalitá single user). Vi sarete resi conto voi stessi del fatto che, nonostante carichiate X (quindi imponete un passaggio al *runlevel* 5) le console siano sempre disponibili e pronte.

Per quanto riguarda il login grafico, basta impostare la seguente linea nel file "inittab" con l'accortezza di impostare il *runlevel* di default a 5.

```
x:5:respawn:/etc/X11/prefdm -nodaemon
```

Si rimanda comunque alle pagine di manuale dei vari processi caricati per avere informazioni più chiare a riguardo.

4.4 Script di avvio

Prima di passare ad una breve analisi degli script di configurazione, illustriamo brevente le modalità di cambio di livello.

4.4.1 Cambiamento del livello di esecuzione

Supponiamo che il livello di esecuzione predefinito sia 3, ma siccome l'utente pippo ama lavorare con X vorremo agevolarlo cambiando il livello di esecuzione a 5. Una possibile strada per farlo è modificare */etc/inittab* e riavviare; dovremmo allora mettere un 5 al posto del 3 alla riga di initdefault oppure, per evitare di riavviare, potremmo semplicemente scrivere da "root":

```
#init 5 <CR>
```

Con questo il sistema passa al livello di esecuzione 5. Naturalmente, al prossimo riavvio il sistema sarà al livello 3 perché col comando *init* il file *inittab* non viene modificato. Nel caso in cui si optasse per la prima modalità (modifica di *inittab*), dato che 6 è il *runlevel* per riavviare, il comando

```
#init 6 <CR>
```

farà riavviare la macchina, mentre

```
#init 0 <CR>
```

farà spegnere la macchina. Eseguire *init* in questo modo può aiutare a comprendere la ragione per cui una delle prime righe di *inittab* contenga l'azione *sysinit*, infatti, quando la macchina è già stata accesa e inizializzata, non è più necessario ripetere l'inizializzazione da capo, e quindi viene semplicemente saltata.

4.4.2 *rc.d/*

Questa directory è costituita da più cartelle che contengono le impostazioni di avvio e chiusura dei demoni per ogni *runlevel* e da una che contiene gli script veri e propri.

Alla prima categoria appartengono le directory *rcN.d* (con N un numero da 0 a 6 ed eventualmene delle lettere "speciali"[13]), mentre alla seconda la sola cartella *init.d*.

Analizziamo ora in maniera più approfondita le cartelle.

[13]Spesso sono dei link ai *runlevel* corrispondenti

Script in *init.d*

La locazione di questa directory (come molte di quelli che seguiranno) dipende dalla distribuzione anche se non è raro trovarla in */etc/rc.d/*. Gli script qui contenuti servono a caricare o rimuovere i *daemon*[14] (demoni o servizi come dir si voglia) con gli opportuni parametri.
Editando ad esempio il file *syslog* si vedrà con che parametri viene caricato e rimosso dalla memoria il demone preposto al logging del sistema. L'analisi di ogni singolo file presente in questa directory sta alla volontà dell'utente che ormai dovrebbe avere buoni strumenti per capire il significato di uno script. Per capire, poi, a che cosa serva un particolare demone basta esaminare la pagina di manuale relativa.

Osservazione: *Si noti che tutti i demoni hanno il nome terminante con la lettera "d" appunto*

rcN.d/

In base al valore di N corrisponde un dato *runlevel*; va da sè che in rc5.d, per esempio, verranno elencati i processi da avviare e terminare quando si passa a runlevel 5. I nomi dei file presenti sono costituiti da una struttura comune:

- Il primo carattere è o S o K
- I due caratteri successivi sono un numero decimale
- Il resto è libero (o quasi...)

La lettera "S" indica che il processo in questione andrà eseguito ogniqualvolta il sistema entrerà il corrispondente *runlevel*; la lettera "K" indica che il processo andrà terminato appena il sistema *lascerà* il *runlevel* in questione. I numeri servono a dare la priorità di esecuzione dello script corrispondente, il resto, invece è il nome del servizio in questione. Per esempio:

```
../rc2.d/S80sendmail
```

Eseguirà *sendmail*[15] con "priorità" 80 (ovvero sarà uno degli ultimi a venire eseguito) all'ingresso del sistema a *runlevel* 2. Al passaggio da un *runlevel* all'altro, *init* confronterà la *kill list*[16] del livello attuale con la *start list* del livello di destinazione e determinerà di conseguenza quali demoni andranno caricati o terminati.

Osservazione: *I file presenti in queste directory sono semplicemente dei link ai corrispondenti script nella cartella* init.d/.
Per intenderci:

```
S80sendmail -> ../init.d/sendmail
```

è un link a init.d/sendmail *che verrà eseguito col paramero* start *qualora si entri nel* runlevel *che contiene il file.*

[14] Un demone è un processo generalmente eseguito in *background* progettato per fornire servizi
[15] Nome del demone MTA (Mail Transport Agent)
[16] la lista dei processi da eliminare all'uscita del *runlevel*

4.5 *BSD* (Non-SysV) init

Distribuzioni come la *Slackware* hanno una configurazione di inizializzazione diversa da quella del *System V*, per cui parte delle indicazioni qua date, non risulta valida. I *BSD* hanno semplicemente una cartella */etc/rc.d/* contente i file di inizializzazione. Gli script sono abbastanza semplici e ben commentati, inoltre i singoli nomi dei file sono spesso autoesplicativi. (ex. *rc.M* corrisponde al *runlevel* di multiuser, *rc.inet1* si occupa di caricare i servizi fondamentali di inetd e così via)

Capitolo 5

Introduzione a VIM

di Elena "of Valhalla" Grandi

5.1 Perché vi(m)

Sotto unix "Tutto è un file", frequentemente di testo; da qui la grande attenzione rivolta verso gli editor relativi in ambiente unix – e quindi anche linux – sia dal punto di vista qualitativo che quantitativo.

Nella massa di editor di testo presenti, alcuni sono molto simili a `notepad` di windows, ne condividono l'estrema semplicita', ma anche la mancanza di numerose funzioni, e per di più richiedono la presenza di X, che non sempre è disponibile, specialmente quando è necessario modificare dei file di configurazione. Questo suggerisce di acquistare una certa familiarità con almeno un'editor di testo da consolle, tra i quali i più diffusi sono emacs e vi (e i suoi cloni).

Emacs [1] è stato uno dei primi programmi open source e comprende davvero tutto, dall'editor di testo al browser, dal lettore di newsgroups ad un linguaggio di programmazione, dal programma per la posta all'assistenza psichiatrica! per contro ha una richiesta di risorse non indifferente, specialmente su macchine non recentissime, e non sempre è disponibile.

Vi invece segue la filosofia del "tanti piccoli programmi che eseguono bene un solo compito", non ha requisiti eccessivi, è presente praticamente su ogni unix ed è disponibile anche per altri sistemi (ad esempio windows); inoltre la sua effettiva osticità verso l'utente è stata parzialmente attenuata dai suoi cloni più recenti, come VIM, che introduce nuove funzioni, ad esempio, l'evidenziazione a colori della sintassi.

Può valere la pena di citare anche il "terzo incomodo" nella "guerra degli editor": ed[2], effettivamente presente sulla gran parte dei sistemi, al pari di vi (quantomeno per quello che riguarda il mondo unix), ma la cui osticità è ancora maggiore di quella del primo impatto con vi, e la cui scomodità per le abitudini

[1] Per i più maligni tra i fan di vi: Emacs Makes A Computer Slow
[2] ed is the standard editor

attuali (è un editor di linea, ovvero che può lavorare solo su una linea per volta) ne fanno una scelta improponibile per l'uso interattivo.

In questa dispensa parleremo di VIM, Vi IMproved, uno dei cloni di vi più diffusi, anche grazie ad un certo numero di migliorie rispetto all'originale, tra cui l'uso del colore per evidenziare la sintassi di numerosi linguaggi. La maggior parte dei comandi indicati comunque fa parte dell'insieme standard e può essere usata anche con gli altri cloni; i casi in cui questo non è possibile verranno segnalati nel testo.

5.2 Le modalità

Per motivi storici, vi utilizza soltanto la parte principale della tastiera[3], per cui necessita di diverse modalità: comando (o normale), inserimento, linea di comando ed altre; per passare dall'una all'altra si usano prevalentemente i seguenti comandi:

- `i` o `a` per passare dalla modalità comando a quella inserimento, partendo dal carattere corrente o da quello successivo rispettivamente;
- `A` per passare dalla modalità comando a quella inserimento, partendo dall'ultimo carattere della linea;
- `R` per passare dalla modalità comando a quella sostituzione;
- `r` per passare dalla modalitá comando a quella sostituzione per la modifica di un solo carattere (dopo torna automaticamente in modalità comando)
- `:` per passare dalla modalità comando a quella linea di comando;
- `/` per passare dalla modalità comando a quella ricerca;
- <Esc> per passare dalle modalità inserimento o sostituzione a quella comando;
- <Backspace> per passare dalla modalità linea di comando a quella comando.

Questo può sembrare una scomodità, ma permette di non spostare mai le mani sulla tastiera, garantendo una velocità di lavoro superiore a quella data da ogni altro editor di testo.

5.3 La modalità riga di comando

In questa modalità si possono inserire comandi (anche abbastanza complessi) che generalmente hanno un effetto di tipo "globale"; ad esempio apertura, salvataggio, chiusura, ricerca eccetera. Entrando in questa modalità, il primo carattere dell'ultima riga, seguito dal cursore, indica il tipo di comando che si può inserire:

- `:` permette di inserire comandi "globali" (detti "comandi Ex");
- `/` o `?` permette di effettuare una ricerca.

[3] quella con i tasti alfanumerici, invio e il tasto esc, sempre presenti su ogni macchina

5.3.1 Principali "comandi Ex"

I comandi "globali" più utilizzati sono quelli per la gestione dei file:

- `:w` per salvare il buffer corrente;
- `:w nomefile` per salvare il buffer corrente con il nome nomefile;
- `:q` per chiudere il buffer corrente ed uscire da vi qualora questo sia l'ultimo (se non sono state fatte modifiche dall'ultimo salvataggio);
- `:q!` per chiudere il buffer corrente, ed eventualmente uscire da vi, senza salvare;
- `:e nomefile` per aprire il file "nomefile" nel buffer corrente;
- `:help` per leggere l'help in linea.

5.3.2 Ricerca e sostituzione

La ricerca di un espressione regolare si effettua semplicemente entrando in modalità riga di comando digitando `/` e quindi digitando la stringa da cercare: il cursore verrà spostato all'inizio di tale stringa e, con VIM, questa verrà evidenziata.

La sostituzione invece avviene tramite il "comando Ex" `:s`, come nel seguente esempio:

```
:%s/old/new/gc
```

oppure

```
:%s/old/new/g
```

dove `%` indica dove sostituire (su tutto il file), `old` è la stringa da sostituire, `new` é la nuova stringa, `g` indica di sostituire tutte le occorrenze e non solo la prima incontrata ed infine l'eventuale `c` fa sìche venga richiesta conferma prima di ogni sostituzione.

5.4 La modalità comando

In modalità comando (o normale) si inseriscono comandi, composti da un solo carattere (o da una combinazione carattere-numero-carattere), che hanno generalmente effetti "locali", sul carattere, parola o riga dove è posto il cursore. I comandi che possono essere applicati ad un solo elemento (carattere, parola, riga, ecc.) sono composti da un solo carattere, quelli che utilizzano la combinazione carattere-numero-carattere agiscono sui [numero] elementi a partire da quello corrente, e possono essere applicati ad un solo elemento omettendo il numero.

5.4.1 Comandi più utilizzati

I comandi "locali" più utilizzati sono:

- `h` per spostare il cursore a sinistra;
- `j` per sostare il cursore in basso;
- `k` per spostare il cursore in alto;
- `l` per spostare il cursore a destra;
- `x` per cancellare il carattere corrente;
- `d[n]d` per tagliare le [n] righe a partire da quella corrente;
- `y[n]y` per copiare le [n] righe a partire da quella corrente;
- `p` per copiare le righe tagliate o copiate sotto quella corrente;
- `u` per annullare l'ultima operazione;
- `.` per ripristinare l'ultimo comando annullato.

5.5 Le modalità inserimento e sostituzione

Queste modalità sono quelle che permettono di scrivere effettivamente il testo: la prima aggiunge i caratteri digitati nella posizione del cursore, la seconda rimpiazza il carattere corrente con quello digitato.

Utilizzando VIM è possibile spostarsi all'interno del testo con i normali tasti (frecce, pag su e pag giù, eccetera), nonché cancellare con il tasto <Backspace>, mentre con vi originale questo non era possibile.

5.6 Per approfondire

Una prima fonte da consultare è la pagina ufficiale di VIM (http://www.vim.org) che contiene anche abbondante documentazione.

VIM dispone inoltre di un help in linea ben nutrito, raggiungibile tramite il comando `:help` o `:help argomento`

Dato l'elevato numero di comandi e il loro difficile reperimento, nel caso si voglia usare vi(m) con una certa sistematicitá può essere il caso di procurarsi una delle numerose "reference cards" con elenchi sintetici dei comandi più utili.

Capitolo 6

Archiviazione, compressione e gestione dei pacchetti

di Elena Grandi Francesco Toso

Introduzione

Da quando esistono i computer si è sempre sentita l'esigenza di "archiviare" (raccogliere più file in uno unico) e comprimere i file, per questioni di comodità nella gestione, di scarsità dello spazio o anche per esigenze dovute a particolari tipi di hardware[1]; quando poi i file da gestire sono quelli di un programma che deve essere distribuito ed installato si aggiunge l'esigenza di automatizzare tale procedura: per questo motivo sono stati inventati i "pachetti" contenenti i file e tutte le informazione necessarie allo scopo.

6.1 Archiviazione e compressione

Contrariamente a ciò che avviene sotto altri sistemi operativi, sotto unix le operazioni di archiviazione e compressione vengono effettuate da programmi distinti, secondo la filosofia del "fai una cosa sola ma falla bene"; questo può sembrare una scomodità, ma presenta alcuni vantaggi dal punto di vista dell'efficienza e dell'aggiornabilità.

6.1.1 `tar(1)`

Il programma tradizionale per l'archiviazione, `tar`, si limita ad accodare ciò che gli viene passato in un'unico file, convenzionalmente dotato dell'estenione `.tar`, aggiungendo le informazioni necessarie per la successiva suddivisione. Il funzionamento di base è estremamente semplice e consiste nei comandi

[1] Ad esempio i nastri, sui quali potrebbe non essere presente un filesystem, ma solo un unico grosso file.

```
$tar cvf filename.tar directory1 [ directory2, ...directoryN ]
```

per archiviare (`c`, Create) le directory da 1 a N nel file (`f`) `filename.tar`, dando informazioni su ciò che sta facendo (`v`, ovvero verbose)

```
$tar tvf filename.tar
```

per visualizzare (`t`, Tree) il contenuto di `filename.tar`, e

```
$tar xvf filename.tar
```

per estrarre (`x`, eXtract) il contenuto di `filename.tar` nella direcotory corrente.

È importante ricordare che quando si archiviano dei files è buona norma inserire nell'archivio anche la directory nella quale sono contenuti, in modo tale che chi compie l'operazione opposta non si trovi con centinaia di file sparsi dove non se li aspettava.

Le nuove versioni di `tar` sono generalmente in grado di gestire file compressi con `gzip` o `bzip2`, aggiungendo rispettivamente `z` o `j`[2] alle opzioni sulla riga di comando: in questo modo `tar` provvede automaticamente a richiamare l'eseguibile opportuno con le dovute opzioni per effettuare la compressione o la decompressione. I file trattati in questo modo possono avere l'estensione `.tar.gz` o `.tar.bz2` rispettivamente, oppure `.tgz`, abbreviazione del primo.

6.1.2 gzip(1)

Il programma di compressione tipico dei sistemi GNU (come Linux) è `gzip`: come il suo predecessore `compress`, si limita alla compressione dei singoli files che gli vengono passati, ai quali aggiunge l'estenzione `.gz`. Il suo utilizzo elementare è semplicemente

```
$ gzip nomefile
```

per comprimere `nomefile` in `nomefile.gz`,

```
$ gzip -d nomefile.gz
```

oppure

```
$ gunzip nomefile.gz
```

per decomprimere `nomefile.gz` in `nomefile`; se gli vengono passati più files si limita a ripetere le operazioni su ciascuno di essi e non può lavorare su directory.

6.1.3 bzip2(1)

`bzip2` è un programma di compressione abbastanza recente, più efficiente di `gzip`, ma non ancora universalmente diffuso: in particolare molti programmi per windows riescono a gestire i file `.gz` e `.tar.gz`, ma non ancora i `bz2` generati da questo programma. L'utilizzo è praticamente identico a quello di `gzip`, utilizzando il comando `bzip nomefile` per la compressione e `bunzip2 nomefile.bz2` per la decompressione.

[2] In alcune versioni di `tar` questa opzione potrebbe essere sostituita da `y`

6.1.4 zip(1)

`zip` è il programma di archiviazione e compressione compatibile con il `PKZIP` per dos e con i vari winzip e simili per windows: il suo utilizzo è generalmente limitato a casi in cui sia necessaria la compatibilità con tali sistemi.

6.1.5 compress(1)

`compress` era il programma standard per la compressione dei file sotto unix: è ormai stato superato per efficacia da `gzip`, che riesce anche a gestirne i file, contraddistinti dall'estensione `.Z`.

6.1.6 cpio(1)

`cpio` è un'altro programma di archiviazione, dotato di alcune funzionalità aggiuntive rispetto a `tar`, ma generalmente poco usato, se non per i pacchetti di tipo rpm (e in questo caso in modo invisibile all'utente).

6.1.7 Ulteriori informazioni

Per saperne di più la cosa migliore è leggere le pagine di manuale (`man`) dei rispettivi programmi.

6.2 I pacchetti rpm

Di Francesco Toso
I pacchetti rpm vennero creati dalla RedHat appositamente per la loro distribuzione con lo scopo di renderla quanto più "user friendly" possibile.
Data la loro semplicitá di utilizzo, vennero ben presto adottati da altre distribuzioni come Mandrake [3] o SuSE. Oltre ai dati veri e propri gli rpm contengono (come tutti i pacchetti) anche dati di controllo come la descrizione del pacchetto, le dipendenze, gli script di installazione e altro.

6.2.1 Struttura

Un pacchetto rpm viene creato basilarmente con la collaborazione di `cpio` e `gzip`; sarebbe quindi possbile, tramite questi due programmi aprire, installare e modificare un pacchetto rpm.
Per una maggior semplicitá d'uso e comodità si consiglia l'uso direttamente di `rpm` con gli opportuni parametri. Tali pacchetti contengono anche un file speciale detto *spec file*. Lo *spec file* contiene la descrizione del software nel pacchetto assieme alle istruzioni su come compilarlo e assieme alla lista dei file binari che verranno installati.

6.2.2 Utilizzo di base

Generalmente `rpm` viene usato per installare pacchetti, ad esmpio:

```
# rpm -i prova-1.0-1.i386.rpm
```

[3] che ricordiamo essere nata come una versione modificata della RedHat

Installerá il pacchetto `prova` versione 1.0-1 compilato per i386 nel sistema (se sono presenti le dovute dipendenze.
Duale all'installazione c'é la disintallazione di un pacchetto:

```
# rpm -e prova
```

Questo comando rimuoverá il pacchetto `prova` informandovi di eventuali problemi di dipendenze con altri pacchetti.
Passiamo ora ad esaminare il contenuto di un dato pacchetto:

```
# rpm -qpi prova-1.0-1.i386.rpm
```

Questo comando mostrerá la descrizione del pacchetto fatta dagli autori; per sapere che file installerá il pacchetto e dove basterá scrivere:

```
# rpm -qpl prova-1.0-1.i386.rpm
```

Per sapere, invece, quali pacchetti sono stati installati sul sistema basterá scrivere

```
# rpm -qa
```

Attenzione: l'output puó essere molto grosso! Si consiglia di ridirigere il flusso su `less` o `more` oppure filtrarlo con `grep`

6.2.3 Funzionalità avanzate

Passiamo alle funzionalitá avanzate di `rpm`. Se volessimo, ad esempio, installare un pacchetto direttamente via `ftp` basterebbe scrivere:

```
# rpm -i ftp://ftp.redhat.com/pub/redhat/rh-2.0-beta/RPMS/\
  foobar-1.0-1.i386.rpm
```

`rpm` stesso si occuperá dello scaricamento e installazione dei pacchetti; tuttavia, a differenza del sistema Debian, non effettuerá né lo scaricamento di dipendenze mancanti, né l'aggiornamento dei pacchetti nel caso in cui risultino datati. É anche possibile semplicemente interrogare il pacchetto remotamente cambiando i comandi come indicato nella sezione precedente.
Supponiamo ora di voler sapere a che pacchetto appartiene il file `iptables` in `/sbin/`:

```
# rpm -qf /sbin/ipchains
```

Avremo cosìil nome del pacchetto che contiene il file; ipotizziamo che tale file sia stato cancellato a nostra insaputa, sarebbe opportuno individuarlo e reinstallare il pacchetto "mutilato":

```
# rpm -Va
```

Interrogherá tutti i pacchetti installati sulla macchina verificando la loro integritá; eventuali mancanze o errori verranno segnalati.

Per installare pacchetti in formato sorgente basta eseguire la seguente operazione:

```
# rpm -bai pacchetto
```

In questo modo si compileranno i sorgenti e si installerá il pacchetto compilato eliminando poi i file temporanei creati. Risulterá molto utile alla fine della sezione seguente.

6.2.4 Creazione

Per poter creare un vero e proprio pacchetto rpm é necessario creare il *file spec* che contiene tutte le informazioni sulla installazione, manutenzione e altro del pacchetto. Si consiglia di chiamarlo seguendo la seguente convenzione: nome-versione-release.spec. Quanto segue é lo scheletro di uno *spec file*

```
Summary: <breve descrizione>
Name: <nome pacchetto>
Version: <versione>
Release: <release>
Copyright: <licenza>
Requires: <dipendenze>
Group: <gruppo del pacchetto>
Source: <url dei sorgenti>
Patch: <eventuale patch applicata>
BuildRoot: <directory di compilazione>
%description
<descrizione per esteso>

%prep
%setup -q
%patch -p1 -b .buildroot

%build
make RPM_OPT_FLAGS="$RPM_OPT_FLAGS"

%install
rm -rf $RPM_BUILD_ROOT
mkdir -p $RPM_BUILD_ROOT/usr/bin
mkdir -p $RPM_BUILD_ROOT/usr/man/man1
install -s -m 755 <nome pacchetto> \
  $RPM_BUILD_ROOT/usr/bin/<nome binario>
install -m 644 <nome pacchetto>.1 \
  $RPM_BUILD_ROOT/usr/man/man1/<nome binario>.1
%clean
rm -rf $RPM_BUILD_ROOT
%files
%defattr(-,root,root)
%doc README TODO COPYING ChangeLog
/usr/bin/<nome binario>
/usr/man/man1/<nome binario>.1
%changelog
<cambiamenti ecc.>
```

Analizziamo ora in dettaglio le varie voci.

Intestazione

Fa parte dell'intestazione tutto ció che viene prima del tag `%prep`. Il significato di ogni campo é abbastanza autoesplicativo. Si segnala la possibilitá di specificare

piú *Source* e *Patch* nel seguente modo (l'esempio é solo per *Source*, per *Patch* basta sostituire la parola chiave):

```
Source0: blah-0.tar.gz
Source1: blah-1.tar.gz
Source2: fooblah.tar.gz
```

Il tag `Group` va compilato secondo quanto specificato in `/usr/doc/rpm*/GROUPS`, si evita di riportare qui di seguito le categorie dato che sono in continuo cambiamento. Ultimo é il tag `BuildRoot` che dá informazioni su dove compilare e installare temporaneamente il pacchetto.

Prep

Qui vengono effettuate tutte le operazioni necessarie per *prep*arare i sorgenti e applicare su questi una o piú *patch*. Tutto ció che viene preceduto dal simbolo `%` é da considerarsi un macrocomando.
Di base `%setup` si limita a decomprimere il contenuto del pacchetto e a entrare nella directory in cui é stato decompresso; alcuni parametri sono:

-n nome imposterá la directory di compilazione. Di default verrá scelta la directory `nome-versione` (attenzione, questa é la fase di compilazione, non di installazione!!)

-b scompatterá la sorgente prima di cambiare directory (risulta assai utile in caso di piú file sorgente)

-D Non cancella la cartella prima di decomprimere. Questa opzione é utile nel caso in cui vi siano piú `%setup` di seguito che collaborano. Si noti che dovrebbe venir usato dal secondo `%setup` in poi ma non nel primo! (altrimenti rischieremmo di avere problemi nella compilazione)

La macro `%patch` si occuperá di modificare i sorgenti in accordo con le *patch* specificate nella sezione principale.

Build

Qui andranno inseriti tutti i comandi per la compilazione dei sorgenti, generalmente é piú che sufficente il comando `make`. La variabile `$RPM_OPT_FLAGS` indica le ottimizzazioni da applicare durante la compilazione e viene impostata come specificato in `/usr/lib/rpm/rpmrc`.

Install

Questa parte riguarda l'installazione vera e propria del pacchetto. Anche qui non vi sono macro per semplificare le operazioni, tuttavia si consiglia di seguire uno schema simile al seguente:

1. Pulizia della BuildRoot tramite `rm - rf $RPM_BUILD_ROOT`
2. Creazione delle directory del programma e della documentazione
3. Installazione vera e propria tramite `install` o `make install`

4. Impostazione dei permessi

Si consiglia di far seguire i comandi di *setup* da uno di `%clean` al fine di eliminare i file superflui a fine installazione.

Files

Qui vanno specificati i file utili creati nella fase di installazione al fine di poterli posizionare correttamente nel file system. In questo caso sono disponibili un paio di macro molto utili:

`%doc` indica quali file di documentazione andranno installati (possono tranquillamente essere messi uno di seguito all'altro); questi verranno posizionati in `/usr/doc/$NAME-$VERSION-$RELEASE`

`%defattr` specifica gli attributi di default per ogni file

`%dir` indica quali directory installare come parte del pacchetto. Nel caso in cui non si specifichi nulla tutto ció che é contenuto nella BuildRoot verrá incluso (quindi verranno installati anche sorgenti puri, file oggetto ecc.)

Una volta creato il file di *spec* non ci resta che costruire il pacchetto vero e proprio. Scelta una directory di lavoro andranno create le seguenti sottodirectory:

`BUILD` dove RPM metterá i compilati

`SOURCES` dove rpm cercherá i sorgenti e le patch per compilare

`SPECS` dove andranno inseriti i file *spec*

`RPMS` dove verranno inseriti i compilati

`SRPMS` dove finiranno i sorgenti scompattati

Posizionati correttamente i file, per testare il tutto basterá scrivere:

```
# rpm -ba nomefile.spec
```

In questo modo si potrá testare la corretta compilazione dell'archivio, inoltre verranno creati i seguenti file:

```
RPMS/i386/*.*.rpm
RPMS/i386/*.*.doc.rpm
SRPMS/*.src.rpm
```

I nomi (gli asterischi) dipendono da molti fattori tra cui il nome del pacchetto, l'architettura su cui si é compilato, la versione e altri.
Ora siete pronti per distribuire i vostri pacchetti!!

6.2.5 Ulteriori informazioni

Si consiglia di usare programmi grafici quali `gnorpm` e `kpackages` in caso di installazione, disinstallazione e semplice gestione dei pacchetti. La loro lentezza é pienamente compensata da una estrema facilitá d'uso. In caso di utilizi avanzati o di necessitá di velocitá il comando da shell risulta il migliore a patto di saperlo utilizzare adeguatamente.
Si consiglia, come sempre, di guardare la pagina di `man` di rpm oltre che l'rpm howto. Una buona guida si puó trovare all'indirizzo:
http://newbie.linuxbe.org/linux/rpm/seneca-rpm.html

6.3 I pacchetti deb

La distribuzione Debian utilizza un suo formato di pacchetti: quelli binari hanno l'estensione `.deb`, inoltre esiste un formato distinto per i pacchetti contenenti i sorgenti, composto da una descrizione (`.dsc`), un archivio compresso (`.orig.tar.gz`) ed un file di differenze (`.diff.gz`).

6.3.1 Struttura

Un pacchetto binario debian è un archivio compresso contenente un file `control` con le informazioni sul pacchetto e sulle dipendenze, un file `confiles` con l'elenco dei file di configurazione, degli script per l'installazione (`preinst` e `postinst`) e per la disinstallazione (`prerm` e `postrm`).

6.3.2 Utilizzo di base

Il sistema di gestione dei pacchetti Debian comprende numerosi programmi, tuttavia nella maggior parte dei casi sono sufficienti il programma interattivo `dselect`, oppure `apt-get`; quest'ultimo richiede un file di configurazione `/etc/apt/sources.list` con l'elenco delle fonti contenenti distribuzioni debian, quindi lo si può usare con:

```
# apt-get update
```

per aggiornare l'elenco locale dei pacchetti disponibili,

```
# apt-get upgrade
```

per aggiornare tutti i pacchetti installati,

```
# apt-get install nomepacchetto
```

per installare o aggiornare un pacchetto specifico, soddisfando le dipendenze.

6.3.3 Ulteriori informazioni

Per una spiegazione approfondita sull'uso dei pacchetti Debian si può consultare "Appunti di informatica libera" di Daniele Giacomini.

6.4 I pacchetti slackware

La distribuzione Slackware utilizza un suo formato di pacchetti, non usato da altre distribuzioni.

6.4.1 Struttura

I pacchetti Slackware sono dei semplici archivi `tar` compressi con `gzip` contenenti i file (generalmente già compilati) del programma ed alcuni script per la gestione dell'installazione:

`/install/doinst.sh` per la creazione dei collegamenti simbolici,

`/var/log/setup/setup.nomepacchetto` per la configurazione del pacchetto in questione,

`/var/log/setup/setup.onlyonce.nomepacchetto` per la configurazione subito dopo l'installazione.

I pacchetti Slackware hanno convenzionalmente l'estensione `.tar.gz` o più frequentemente `.tgz` caratteristica dei programmi usati per l'archiviazione e la compressione.

6.4.2 Utilizzo di base

Il modo più intuitivo per gestire i pacchetti slackware è l'utilizzo del programma `pkgtool(8)`, che permette di effettuare l'installazione, la disinstallazione e il controllo dei pacchetti attraverso un'interfaccia in grafica ascii; questo programma tuttavia è solo un front-end per `installpkg(8)` e `removepkg(8)`.

Per l'installazione di un pacchetto è sufficiente il comando

```
# installpkg nomepacchetto
```

(notare il prompt di root), mentre per la sua disinstallazione si usa

```
# removepkg nomepacchetto
```

in entrambi i casi l'opzione `-warn` permette di simulare soltanto l'operazione richiesta.

6.4.3 Funzionalità avanzate

Per l'aggiornamento dei pacchetti è possibile usare il comando

```
# upgradepkg vecchiopacchetto[%nuovopacchetto]
```

che installa `nuovopacchetto` e quindi rimuove i files di `vecchiopacchetto` non richesti da quello nuovo.

Qualora si vogliano installare i pacchetti Slackware non a partire dalla root "standard" (`/`), ma da una directory a piacere, ad esempio perchè si sta preparando un'installazione "from scratch" oppure per un'altra macchina, è possibile utilizzare la variabile `ROOT`, impostata con il nome della directory dalla quale si vuol far partire il "nuovo" filesystem; per tale scopo, ma solo per il comando `installpkg`, è anche disponibile l'opzione `-root /nomenuovaroot`.

6.4.4 Creazione

Per creare un pacchetto Slackware con il contenuto della directory corrente e di tutte le sue subdirectory si utilizza il comando

```
# makepkg nomepacchetto
```

che crea un'archivio nel quale però i link simbolici sono stati eliminati e sono state aggiunte[4] le istruzioni su come ricrearli nello script `install/doinst.sh`;

[4] aggiunte in coda – in inglese appended – al file se questo è già presente, altrimenti viene creato.

eventuali comandi da eseguirsi in fase di installazione e/o configurazione possono essere messi in tale script o in quelli presenti in `var/log/setup`, utilizzando la sintassi della shell Bourne (in alcuni casi in fase di installazione verranno eseguiti usando la shell `ash`!), avendo cura di non mettere nessun comando interattivo in `install/doinst.sh`.

6.4.5 Ulteriori informazioni

Per ulteriori informazioni si possono leggere le seguenti pagine di manuale: `installpkg(8)`, `explodepkg(8)`, `removepkg(8)`, `upgradepkg(8)`, `makepkg(8)`, `setup(8)` e `pkgtool(8)`, ed eventualmente per la creazione degli script `ash(1)`.

6.5 I pacchetti con i sorgenti

Il formato tradizionale per la distribuzione dei programmi sotto unix, e in alcuni casi il migliore, è il codice sorgente, generalmente raccolto in archivi compressi insieme a script ed altri file utili per la compilazione.

6.5.1 Struttura

Questi pacchetti sono dei normali archivi (di solito `tar`) compressi con uno dei programmi di compressione (di solito `gzip` o `bzip2`), contenenti il codice sorgente, dei file di testo di presentazione del programma o di aiuto (`README`, `INSTALL`) ed alcuni file utili per la compilazione, come lo script `./configure` e il *makefile*, di solito chiamato `makefile` o `Makefile` e spesso generato da `./configure`.

6.5.2 Utilizzo di base

Quando si scarica un pacchetto di questo tipo, la prima cosa da fare è estrarne i contenuti in una directory opportuna (una subdirectory della propria home può andare bene se si ha intenzione di cancellare i sorgenti, se si ha intenzione di lasciarli sul disco di solito si utilizza una subdirectory di `/usr/src`). A questo punto è sempre utile leggere i file `README` e/o `INSTALL`, che dovrebbero contenere istruzioni dettagliate per l'installazione; nel caso in cui questi non fossero presenti o fossero lacunosi, la procedura standard prevede comunque l'esecuzione dello script `./configure`, da richiamare sempre indicandone esplicitamente la posizione:

```
$ ./configure [eventuali opzioni documentate nel README]
```

che provvede a generare il *makefile*; nel caso in cui tale script non fosse presente può essere necessario modificare a mano il *makefile* originale, in modo tale da dare al programma `make` le istruzioni adatte al proprio sistema. Quindi si può usare il comando

```
$ make
```

per procedere alla compilazione del programma.

Completati questi passaggi, se il "makefile" li supporta, è possibile usare il comando

```
# make install
```

(come utente root!) per copiare i file generati nelle posizioni opportune (secondo l'autore del "makefile", e quindi il comando

```
$ make clean
```

per eliminare i file oggetto generati nel corso della compilazione.

6.6 Altri tipi di pacchetti

Alcuni programmi sono, per vari motivi, distribuiti in pacchetti di tipo diverso da quelli "standard".

6.6.1 Eseguibili installanti

Sono file eseguibili che forniscono una procedura di installazione simile a quella tipica di windows, con un'interfaccia grafica che chiede alcune domande e quindi procede all'installazione. Possono essere distribuiti da soli, oppure archiviati e compressi con i programmi standard, eventualmente insieme a semplici file di aiuto (`README`). Non sono molto diffusi: da un lato perche' non esistono programmi "standard" per la loro realizzazione, come avviene invece sotto windows, dall'altro perchè non offrono le funzionalità di controllo delle dipendenze e supporto per la disinstallazione fornite dai pacchetti specifici per le distribuzioni.

6.6.2 Net installer

Sono dei file eseguibili di piccole dimensioni che forniscono un'interfaccia simile a quella degli installer precedenti per la configurazione, quindi provvedono a scaricare da internet le sole parti del programma delle quali si è richiesta l'installazione.

6.6.3 pacchetti jar

I pacchetti `jar` contengono file oggetto java archiviati e compressi e possono essere usati direttamente dalla java virtual machine, generalmente con il comando `$ java -jar nomepacchetto.jar`.

6.7 Glossario

Glossario dei termini tecnici usati nella dispensa.

archiviazione operazione di raccolta di più files in uno unico, detto archivio, comprendente anche le informazioni necessarie per la reversibilità del processo.

compilazione operazione che trasforma del codice sorgente scritto in un linguaggio ad alto livello in codice macchina eseguibile.

compressione operazione di riduzione dello spazio occupato da un file, effettuata mediante l'applicazione di un apposito algoritmo matematico.

dipendenze si chiamano dipendenze di un pacchetto tutti e soli i pacchetti che siano strettamente necessari per il corretto funzionamento dello stesso.

pacchetto mezzo di distribuzione di un programma, comprendente i file dello stesso ed alcuni file per la sua gestione. Nella maggior parte dei casi si può tranquillamente identificare il pacchetto con l'archivio che lo contiene; fa eccezione la distribuzione Debian, nella quale le due cose sono da considerarsi ben distinte.

sorgenti (o codice sorgente) istruzioni di un programma scritte in un linguaggio di alto livello.

6.8 La struttura convenzionale del filesystem

Nei sitstemi unix è frequentemente presente una struttura di directory convenzionali, per dare un ordine alle miriadi di file presenti: è utile conoscerla[5] per poter gestire al meglio il proprio sistema, specialmente in fase di installazione di software.

`/bin` contiene gli eseguibili dei programmi fondamentali per il funzionamento del sistema (ad esempio `sh` o `cd`);

`/sbin` contiene gli eseguibili dei programmi fondamentali per il funzionamento e la gestione del sistema, che debbano essere usati solo dall'utente root (Superuser BINaries);

`/etc` contiene i file di configurazione dei programmi contenuti in `/bin` e `/sbin`;

`/lib` contiene le librerie fondamentali;

`/opt` contiene le directory relative a programmi "opzionali", che richiedono una propria directory;

`/var` contiene i files "variabili" dei programmi in `/bin` e `/sbin`, come i log eccetera;

`/usr` contiene i file necessari agli utenti, come i programmi aggiuntivi (suddivisi in `/usr/bin`, `/usr/etc` `/usr/sbin`;

`/boot` contiene alcuni file necessari per effettuare il boot del sistema (tra cui in alcuni casi anche il kernel);

`/dev` contiene i file di "device" corrispondenti alle varie periferiche del sistema;

`/home` contiene le directory degli utenti (`/home/nome`);

`/mnt` contiene eventuali punti di mount di partizioni e/o dischi non montati altrove (ad es. `/mnt/cdrom`);

`/root` contiene la directory home dell'utente root;

`/tmp` contiene eventuali file temporanei.

[5] almeno nella variante utilizzata dalle principali distribuzioni linux

Capitolo 7

Introduzione alla sicurezza dei sistemi informativi

di Francesco Toso

Questo documento e l'autore non hanno alcuna pretesa di spigare in maniera esaustiva le problematiche sulla sicurezza dei sistemi informativi. Per poter trattare efficacemente questo argomento non basterebbe un libro intero e trenta ore di lezione "seria"; quanto segue vuole solo essere una panoramica sullo stato attuale della sicurezza informatica.

7.1 Definizione di sicurezza in ambito informatico

Un sistema informatico sicuro anzitutto deve garantire l'*integrità* e la *confidenzialità* delle informazioni in esso contenute e fornite; questo significa che le informazioni non devono poter essere manomesse e devono essere accessibili solo ed esclusivamente ai destinatari voluti.

In una azienda farmaceutica, ad esempio, non si vuole che un qualsiasi operatore dell'azienda possa modificare le composizioni di un medicinale in maniera arbitraria, inoltre solo chi è nel reparto di ricerca potrà avere accesso alle formule relative al proprio lavoro.

Queste esigenze non sono nate negli ultimi anni come si potrebbe pensare ma hanno origini assai più vecchie: in ambito militare, infatti,si è sempre cercato di far giungere le proprie strategie belliche al fronte senza che gli avversari intercettassero il messaggio e in modo che non potessere essere né letto né modificato da un eventuale intercettatore.[1]

Purtroppo, dato che si ha a che fare con del software spesso mal progettato dal punto di vista della sicurezza, attualmente un sistema informatico sicuro è

[1] Si ricorda il crittosistema di Cesare

un sistema che che garantisce un elevato grado di integrità e confidenzialità dei dati riducendo al minimo le probabilità di intrusione da parte di estranei. Si ricordi che in un sistema informatico il livello totale di sicurezza del sistema è dato da quello della macchina più debole!

Per ottenere quanto scritto è improponibile fare affidamento *solo* su del software che automatizzi i processi, è necessaria la presenza di uomini che costantemente si tengano aggiornati e che non "abbassino mai la guardia".

7.2 Caratterizzazione degli attaccanti

Ma chi sono i "nemici"? In generale quando ci si deve difendere è bene conoscere al meglio i propri nemici in modo da sfruttare le loro debolezze a nostro vantaggio. Difendersi da ignoti è molto più complicato e oneroso e questo è uno dei principali problemi in informatica: difficilmente si vedrà in faccia o si conoscerà chi ha sfondato il nostro sistema e molto spesso ci si accorgerà quando ormai il danno è stato fatto. Questo grosso svantaggio per gli amministratori di sistema ha portato a generare una catalogazione comportamentale dei nemici al fine di ridurre il fattore di incertezza. Inutile elencare tutte le diatribe nate in merito; tuttavia parte di questa classificazione ha dato la possibilità (possbilià assai poco sfruttata) agli amministratori di difendersi da una buona fetta dei "nemici".

I cosiddetti *script kiddy*, infatti, molto spesso hanno pochissime nozioni informatiche e riescono a entrare nei sistemi solo grazie a degli script di cui ignorano il contenuto e il funzionamento. Basta quindi cambiare qualche parametro come la posizione di un file critico, il nome o altro per mettere in seria difficoltà questa classe di persone (che spesso è una buona fetta dei "nemici") e farli desistere.

La restante parte solitamente è molto motiviata e difficilmente si fermerà davanti a problemi anche di grossa entità (spesso un *hacker* ha dalla sua il tempo).

7.3 Tipologie di attacco

Effetuiamo ora una tassonomia delle modalità di attacco al sistema per poter megli capire anche dove difendersi.

7.3.1 Dall'esterno della rete locale

Un primo tipo di attacco (di gran lunga il più diffuso) consiste nello sfruttare vulnerabilità del sistema vittima connettendosi direttamente (o quasi) ad esso.

Per prevenire tali attacchi è utile se non obbligatorio aggiornare il software installato ogni qual volta si scopre che la versione attualmente in uso ha dei problemi di sicurezza. Risulta inoltre indispensabile non installare servizi in versione beta o "troppo avanti" nelle versioni.[2] La corretta installazione di un firewall e l'utilizzo di uno dei paradigmi descritti in seguito limita notevolmente i danni causabili al sistema.

[2]Si veda qmail contro sendmail

7.3.2 Man in the middle

Letteralmente *Uomo nel mezzo*, è il tipo di attacco più comune e frequente in ambito militare. Spesso, infatti, non serve introdursi sul server ma basta semplicemente "spiare" i dati trasmessi dal server ai client per ottenere informazioni preziose quali password, numeri di carte di credito ed altro.

Il sistema più efficace per questo tipo di attacco è quello di adottare trasmissioni crittate (eventualmente solo quando c'è necessità) o addirittura ambienti completi come Kerberos.

7.3.3 Interno della rete

Come si dovrebbe imparare dall'Iliade, il pericolo peggiore spesso proviene dall'interno. Le classiche reti NT/9x sono delle mine vaganti per la sicurezza. L'utente di client 9x, infatti, lascia spesso e inavvertitamente aperte porte netbios per la condivisione di file su cartelle "critiche" permettendo a chiunque l'accesso[3].

Se si ricorda qualche anno fa si parlò di una intrusione nei sistemi Microsoft; questo fu possibile perché qualcuno lasciò aperta la porta netbios[4] permettendo una facilissima intrusione nel sistema. Non è anche da escludersi che qualcuno del personale voglia appropriarsi di dati o informazioni, per questo risulta necessario garantire un certo livello fisico di sicurezza limitando l'accesso ai locali server e dando in dotazione al personale computer pensati appositamente a tale scopo.

7.3.4 Sé stessi

Per concludere si vuol far presente che spesso sono gli stessi amministratori (o presunti tali) che rendono la rete insicura e vittima di attacchi. La pigrizia di queste persone nel non volere aggiornare il software presente, il fatto di dare password e account liberamente e la mancanza di voglia di cambiare periodicamente la propria password, fanno si che anche i più sprovveduti riescano a sfondare il sistema.

L'unico "antidoto" contro questo problema (assai frequente) è quello di far venire la voglia di lavorare all'amministratore; purtroppo non esistono mezzi scientifici per porvi rimedio.

7.4 Tecniche di difesa

Dopo aver visto a grandi linee le modalità di attacco passiamo alla descrizione delle modalità *passive* di difesa. Con *difesa passiva* si indicano quegli accorgimenti che permettono di aumentare la sicurezza del sistema informativo, che non riguardano direttamente il software installato sul sistema e che sono "sempre attivi".

[3] Il protocollo netbios, infatti, non si preoccupa minimamente di autenticare chi si vuole collegare e non implementa alcun meccanismo di restrizione degli accessi.

[4] In questo caso risulta stupido anche l'amministratore che al tempo non si preoccupò minimamente di controllare la rete

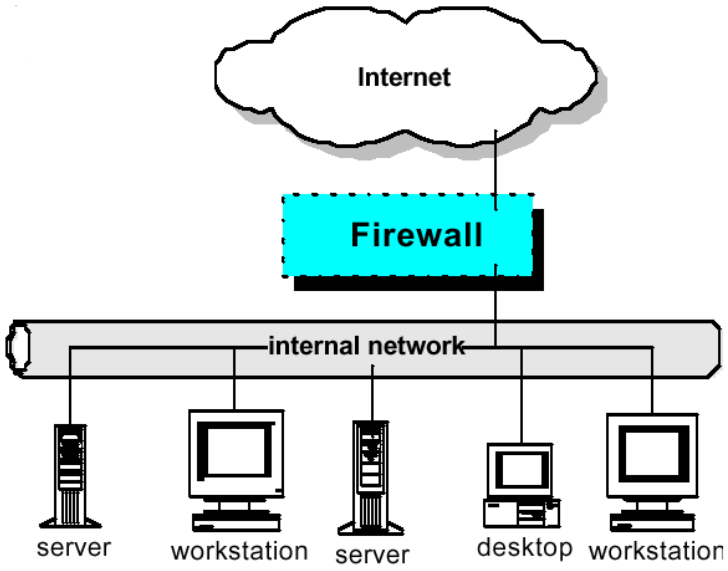

Figura 7.1: Firewall semplice (G. Serazzi, Cremonesi)

7.4.1 Policy di sicurezza

Un primo sistema di *difesa passiva* che non è assolutamente opzionale per gli ammninistratori è quello di definire una *policy*[5] per la sicurezza del sistema informativo come ad esmpio le modalità di cessione degli account o gli accessi al sistema. Più è restrittivo (e rispettato) questo insieme di regole, maggiore è la sicurezza del sistema; risulta evidente che è necessario trovare un buon compromesso.

Come ogni attività progettuale ben fatta, richiede un'analisi preliminare del sistema informativo e delle esigenze degli utenti cosìcome una attenta stesura della policy stessa in maniera tale che possa essere facilmente modificata in seguito. Un fattore importante è la segretezza della stessa: allo stato attuale delle cose meno persone conoscono la struttura vera e propria della policy, minori risultano le possibilità di intrusione nella rete dall'interno.

7.4.2 Paradigmi di sicurezza

Si elencano ora una serie di paradigmi di configurazione della rete al fine di arginare il più possibile il danno in caso di intrusione nel sistema.

Firewall semplice

Questa struttura permette di filtrare ogni pacchetto proveniente dall'esterno ed eventualmente bloccarlo secondo certi criteri. Tale configurazione viene spesso adottata quando non si vuole esportare verso l'esterno alcun servizio se non a un insieme di macchine autorizzate.

[5] Un insieme di regole

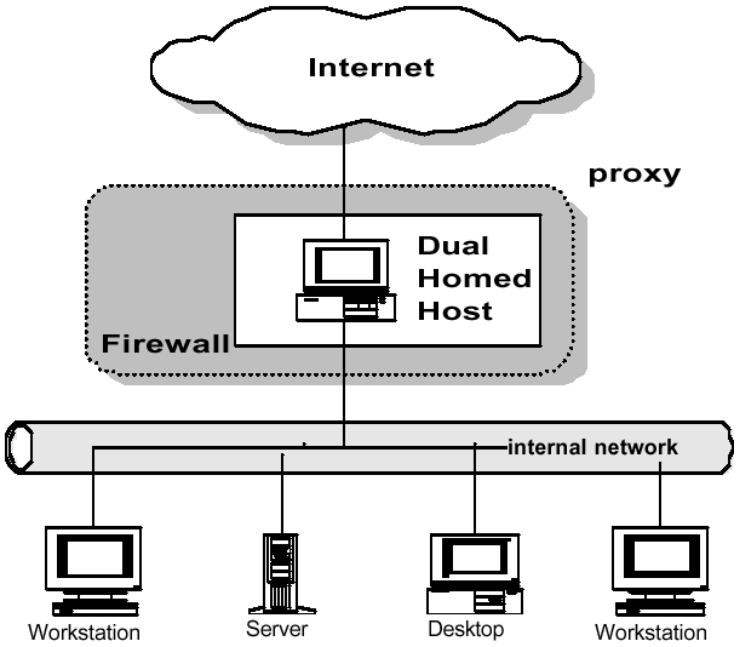

Figura 7.2: Proxy masquerading (G. Serazzi, Cremonesi)

Ha il grosso svantaggio che, se viene in un qualche modo sorpassato il firewall, tutto il resto è facilmente visibile, inoltre non difende da attacchi provenienti dall'interno o anche da porte aperte da virus o simili.

Proxy masquerading

Il proxy ha due interfacce di rete: una per la rete interna, l'altra per l'esterna. In questo modo si dividono nettamente le due reti: da quella esterna non si può accedere all'interno e viceversa se non tramite *filtraggio* del proxy.

Bastion host con packet filtering a due livelli

Si conclude con questo paradigma tanto complicato quanto efficace che permette alle aziende di esportare servizi come ftp o http direttamente dalla rete locale senza (o comunque riducendo al minimo) esporre la rete interna a problemi di intrusione e di accesso non autorizzato ai dati.

I cosiddetti bastion host, infatti, contengono solo ed esclusivamente informazioni che possono essere rese pubbliche senza problemi, nascondendo oltre il secondo firewall i dati più sensibili.

7.5 Bibliografia

1. Huges Larry Jr., *Tecniche di sicurezza internet* Jackson Libri

2. Security HOWTO

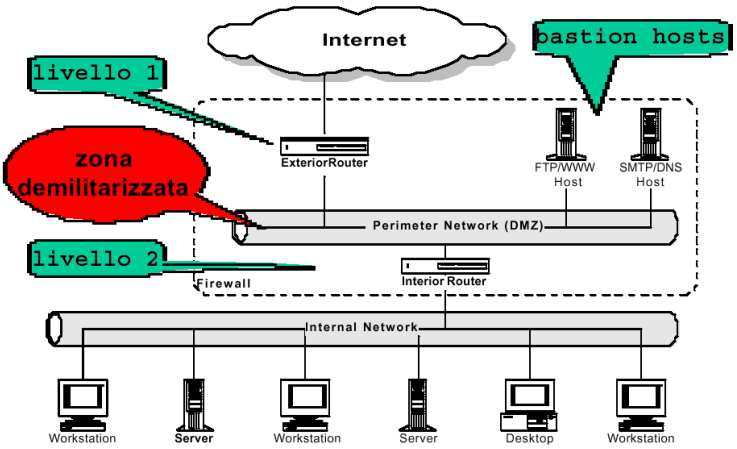

Figura 7.3: Bastion host con packet filtering a due livelli (G. Serazzi, Cremonesi)

3. Net HOWTO

4. prof. Serazzi & dott. Cremonesi, *Dispense sulla sicurezza*