La struttura di un sistema GNU/Linux.

Simone Piccardi

8 agosto 2002

Indice

1 L'architettura di base.

Contrariamente ad altri sistemi operativi, GNU/Linux nasce, come tutti gli Unix, come sistema multitasking e multiutente. Questo significa che GNU/Linux ha una architettura di sistema che è stata pensata fin dall'inizio per l'uso contemporaneo da parte di più utenti, cosa che comporta conseguenze non del tutto intuitive nel caso in cui, come oggi sempre più spesso accade, esso venga usato come stazione di lavoro da un utente singolo.

Il concetto alla base di ogni sistema Unix come GNU/Linux è quello di avere una rigida separazione fra il cosiddetto *kernel* (il *nucleo* del sistema) a cui si demanda la gestione delle risorse hardware come la CPU, la memoria e le periferiche e tutto il resto del sistema, che comprende i comandi base, gli applicativi, le interfacce per l'interazione con gli utenti; tutti questi verranno realizzati con appositi programmi eseguiti dal kernel che vengono chiamati *processi*.

Lo scopo del kernel infatti è solo quello di essere in grado di eseguire contemporaneamente molti processi in maniera efficiente, garantendo una corretta distribuzione fra gli stessi della memoria e del tempo di CPU, e quello di provvedere le adeguate interfacce software per l'accesso alle periferiche della macchina e le infrastrutture di base necessarie per costruire i servizi. Tutto il resto, dall'autenticazione all'interfaccia utente, è demandato all'esecuzione di opportuni processi.

Questo si traduce in una delle caratteristiche essenziali su cui si basa l'architettura dei sistemi Unix: la distinzione fra il cosiddetto *user space*, che è l'ambiente a disposizione degli utenti, in cui vengono eseguiti i processi, e il *kernel space*, che è l'ambiente in cui viene eseguito il kernel. I due ambienti comunicano attraverso un insieme di interfacce ben definite e standardizzate; secondo una struttura come quella mostrata in fig. 1.

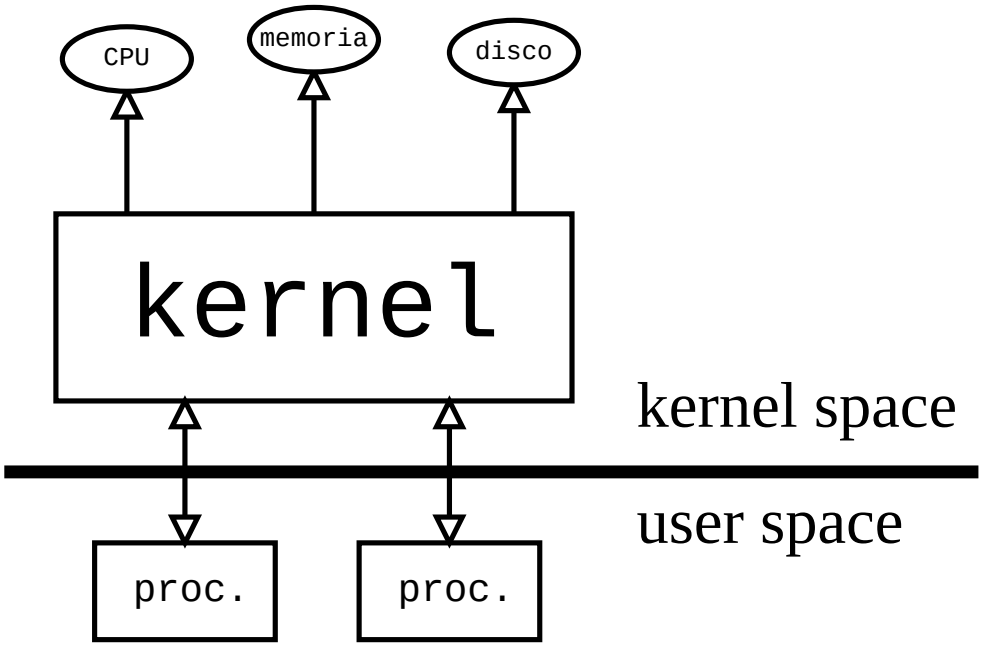

Figura 1: Struttura del sistema

Questa architettura comporta che solo il kernel viene eseguito in modalità privilegiata, ed è l'unico a poter accedere direttamente alle risorse dell'hardware; i normali programmi invece verranno eseguiti in modalità protetta, in un ambiente virtuale (l'*user space* appunto) in cui essi si vedono come se avessero piena disponibilità della CPU e della memoria.

Sarà sempre il kernel ad eseguire al suo interno le operazioni di accesso alle risorse richieste dai vari processi, e a decidere volta per volta qual'è il processo che deve essere eseguito, realizzando così il multitasking, il tutto in modo sostanzialmente trasparente ai processi stessi.

Una conseguenza di questa separazione è che non è possibile ad un singolo programma disturbare l'azione di un altro programma o del kernel stesso, e questo è il principale motivo della stabilità di un sistema Unix nei confronti di altri sistemi in cui i processi non hanno di questi limiti, o vengono, per vari motivi, eseguiti all'interno del kernel.

Per illustrare meglio la distinzione fra kernel space e user space prendiamo in esame la procedura di avvio del sistema. All'accensione del computer viene eseguito il programma che sta nel BIOS; questo dopo aver fatto i suoi controlli interni esegue la procedura di avvio del sistema. Nei PC tutto ciò viene effettuato caricando dal dispositivo indicato nei settaggi del BIOS stesso un apposito programma, il *bootloader*, (i due più usati per Linux sono `lilo` e `grub`) che a sua volta recupera (in genere dal disco) una immagine del kernel che mette in memoria per poi farne partire l'esecuzione.

Una volta che il controllo è passato al kernel questo si incaricherà di montare (vedi sez. 5) il disco di boot, e farà partire il primo processo. Per convenzione questo processo si chiama `init`, ed è il programma di inizializzazione che a sua volta farà partire tutti gli altri processi che permettono di usare il sistema.

Fra questi processi ci sarà anche quello che si occupa di dialogare con la tastiera e lo schermo della console, quello che chiede nome e password dell'utente che si vuole collegare, e quello che una volta completato il collegamento (procedura che viene chiamata *login*) mette a disposizione dell'utente l'interfaccia da cui inviare i comandi (sia questa una shell a riga di comando o una interfaccia grafica).

È da rimarcare come tutti i comandi di base di un sistema GNU/Linux, come quelli per vedere la lista dei file, o per entrare nel sistema, non abbiano niente a che fare con il kernel, ma siano dei semplici programmi che vengono eseguiti dal kernel e fanno operazioni attraverso le funzioni (dette *system call*) che esso mette a disposizione, esattamente come accadrebbe per un programma di scrittura o di disegno.

Questo significa ad esempio che il kernel di per sé non dispone di primitive per tutta una serie di operazioni (come la copia di un file) che altri sistemi operativi (come Windows) hanno al loro interno: tutte le operazioni di normale amministrazione di un sistema sono sempre realizzate da normali programmi.

2 Il funzionamento del sistema

Benché costituisca il cuore del sistema, il kernel da solo sarebbe assolutamente inutile, così come sarebbe inutile da solo il motore di una automobile, senza avere le ruote, lo sterzo, la carrozzeria, e tutto il resto. Per avere un sistema funzionante infatti occorre avere, oltre al kernel, anche tutti i programmi che permettano all'utente di eseguire le varie operazioni con i dischi, i file, le periferiche.

Per questo al kernel vengono sempre uniti degli opportuni programmi di gestione ed è l'insieme di questi e del kernel che costituisce un sistema funzionante. Di solito i rivenditori (o anche gruppi di volontari, come nel caso di Debian) si preoccupano di raccogliere in forma coerente i programmi necessari, per andare a costruire quella che viene chiamata una *distribuzione*. Sono in genere queste distribuzioni (come Debian, Mandrake, RedHat, Slackware[1]), quelle che si trovano sui CD con i quali si installa "Linux".

Il gruppo principale di questi programmi, e le librerie di base che essi e tutti gli altri programmi usano, derivano dal progetto GNU della Free Software Foundation: è su di essi che ogni altro programma è basato, ed è per questo che è più corretto riferirsi all'intero sistema come a GNU/Linux, dato che Linux indica solo una parte, sia pure fondamentale.

Anche se il kernel tratta tutti i programmi allo modo, non tutti hanno la stessa importanza.

[1] in rigoroso ordine alfabetico!

Nella sezione precedente ad esempio abbiamo già incontrato uno programma particolare, `init`, che è quello che si cura dell'inizializzazione del sistema quando questo viene fatto partire.

Benchè in teoria sia possibile far partire qualunque altro programma al posto di `init`[2] tutti i sistemi Unix usano questo programma come primo processo lanciato all'avvio del sistema, per provvedere a mettere in esecuzione tutti gli altri programmi che serviranno ad avere un sistema funzionante con cui lavorare.

Così a seconda dei programmi che `init` mette in esecuzione (che a loro volta potranno lanciarne di altri) ci si può trovare davanti ad un terminale a caratteri o ad una interfaccia grafica, e a seconda di quanto deciso dall'amministratore (di norma lo si fa in fase di installazione, ma lo si può cambiare anche in seguito) si avrà un server di posta, o un server web, ecc.

Questo è possibile perché delle caratteristiche fondamentali di Unix è che qualunque processo può a sua volta lanciarne degli altri, che vengono detti *processi figli*. Il processo originale è detto *padre*, e tutti i processi, eccetto `init` che viene lanciato dal kernel all'avvio, hanno un padre. Questo permette di organizzare i processi in una gerarchia, e ci mostra come `init` venga a ricoprire nel sistema un ruolo del tutto speciale.

Un elenco dei processi attivi del sistema può essere ottenuto con il programma `ps`, questo, con le opzioni specificate, permette di vedere l'elenco in forma gerarchica evidenziando la relazione padre/figlio[3]:

```
[piccardi@hogen piccardi]$ ps axf
  PID TTY      STAT   TIME COMMAND
    6 ?        SW     0:00 [kupdated]
    5 ?        SW     0:00 [bdflush]
    4 ?        SW     0:00 [kswapd]
    3 ?        SWN    0:00 [ksoftirqd_CPU0]
    1 ?        S      0:03 init [2]
    2 ?        SW     0:00 [keventd]
    7 ?        SW     0:00 [kjournald]
   76 ?        SW     0:00 [kjournald]
  106 ?        S      0:00 /sbin/portmap
  168 ?        S      0:00 /sbin/syslogd
  171 ?        S      0:00 /sbin/klogd
  176 ?        S      0:00 /usr/sbin/named
  180 ?        S      0:00 /sbin/rpc.statd
  327 ?        S      0:00 /usr/sbin/gpm -m /dev/psaux -t ps2
  332 ?        S      0:00 /usr/sbin/inetd
  344 ?        S      0:00 lpd Waiting
  435 ?        S      0:00 /usr/lib/postfix/master
  437 ?        S      0:00  \_ pickup -l -t fifo -c
  438 ?        S      0:00  \_ qmgr -l -t fifo -u -c
  448 ?        S      0:00 /usr/sbin/sshd
  908 ?        S      0:00  \_ /usr/sbin/sshd
  909 pts/0    S      0:00      \_ -bash
  919 pts/0    R      0:00          \_ ps axf
  474 ?        S      0:00 /usr/sbin/atd
  477 ?        S      0:00 /usr/sbin/cron
  484 tty2     S      0:00 /sbin/getty 38400 tty2
  485 tty3     S      0:00 /sbin/getty 38400 tty3
```

[2] ed in casi di emergenza si può lanciare una shell al suo posto, o per usi particolare, ad esempio in sistemi embedded che devono svolgere un solo compito, un altro programma specifico

[3] una vista più organica, coi soli nomi dei processi, può essere invece ottenuta con `pstree`

```
  486 tty4     S      0:00 /sbin/getty 38400 tty4
  487 tty5     S      0:00 /sbin/getty 38400 tty5
  488 tty6     S      0:00 /sbin/getty 38400 tty6
  635 ?        SN     0:00 /usr/sbin/junkbuster /etc/junkbuster/config
  672 ?        SN     0:00 /usr/sbin/wwwoffled -c /etc/wwwoffle/wwwoffle.conf
  907 tty1     S      0:00 /sbin/getty 38400 tty1
```

L'output del comando ci da molte informazioni (e con opzioni diverse ce ne potrebbe fornire ancora di più, ad esempio non si è riportato l'utente cui il processo appartiene). In questo caso si è scelto di visualizzare le più importanti. Possiamo vedere che ad ogni processo è associato un numero identificativo, il PID (riportato in prima colonna), ed un terminale di riferimento (riportato in seconda colonna); seguono nella lista lo stato del processo, il tempo di attività ed la stringa di comando usata per lanciare il programma.

Queste poche righe ci permettono di introdurre una distinzione importante fra i vari processi, quella fra processi interattivi e non. Come si può notare infatti molti processi non hanno alcun terminale di riferimento (la seconda riga è un `?`). Questi processi sono tradizionalmente chiamati *demoni*, e sono utilizzati dal sistema per compiere una serie di compiti di utilità, (come spedire la posta, eseguire lavori periodici, servire pagine web, ecc.) anche quando nessun utente è collegato ad un terminale. Per questo lavorano come suol dirsi "in *background*" e non hanno nessun terminale di riferimento.

Altri programmi invece, come la *shell* (che è l'interfaccia a linea di comando) sono usati in modalità interattiva: nell'esempio possiamo vedere come essa (nel caso specifico la shell usata è la `bash`, che è la shell standard di Linux), sia stata appunto usata per lanciare il comando `ps`.

Dall'output del comando (che si è ottenuto dopo il login su una macchina remota su cui non c'erano altri utenti) si vede anche come gli unici altri processi interattivi siano le varie istanze di `getty` (il programma che attiva le console sul terminale, da cui poi si può effettuare il login). Qualora il comando fosse stato lanciato sulla macchina su cui sono state scritte queste dispense si sarebbe avuto un qualcosa del tipo:

```
[piccardi@hogen piccardi]$ ps axf
 PID TTY      STAT   TIME COMMAND
 533 ?        S      0:00 /usr/bin/kdm
 543 ?        S<     0:46  \_ /usr/X11R6/bin/X -dpi 100 -nolisten tcp vt7 -auth
 544 ?        S      0:00  \_ -:0
 560 ?        S      0:01      \_ /usr/bin/WindowMaker
 595 ?        S      0:00          \_ /usr/bin/ssh-agent sh /home/piccardi/.xse
 601 ?        S      0:00          \_ wterm -bg black -fg LightSteelBlue
 616 pts/1    S      0:00          |   \_ bash
1105 pts/1    S      0:00          |       \_ bash
 602 ?        S      0:00          \_ wterm -bg black -fg LightSteelBlue
 615 pts/0    S      0:00          |   \_ bash
 603 ?        S      0:00          \_ wterm -bg black -fg LightSteelBlue
 617 pts/2    S      0:00          |   \_ bash
1395 pts/2    S      0:35          |       \_ emacs struttura.tex
1398 ?        S      0:01          |       |   \_ /usr/bin/ispell -a -m -B
1407 pts/3    S      0:00          |       |   \_ /bin/sh /usr/bin/xdvi struttu
1410 pts/3    S      0:00          |       |       \_ xdvi.bin -name xdvi strut
1411 pts/3    S      0:02          |       |           \_ gs -sDEVICE=x11 -dNOP
1514 pts/2    R      0:00          |       \_ ps -e f
```

con molti altri processi interattivi derivanti dai vari terminali avviati dall'interfaccia grafica.

Tutti gli altri processi mostrati nell'esempio precedente eseguono dei demoni, che possono essere in qualche modo pensati come processi di sistema, anche se, come detto all'inizio, dal punto di vista del kernel questi sono processi come tutti gli altri.

Esempi di demoni sono `sshd`, che gestisce le connessioni criptate da altri computer (il quale, come si vede, ha creato un figlio che esegue la shell da cui abbiamo dato il comando), `postfix` che si occupa dell'inoltro della posta elettronica (e ha generato due altri processi per eseguire il compito), `syslogd` che genera i file di log del sistema, ecc.

In generale il sistema, a seconda della configurazione, fa partire i demoni prescelti dall'amministratore; gran parte delle distribuzioni di Linux (tutte eccetto Slackware) usano quello che si chiama avvio in modalità System V (torneremo su questo nella lezione sull'amministrazione di base); questo permette, attraverso l'uso di opportuni script di shell di specificare dei *run level*, cioè una serie di *modalità operative* del sistema in cui vengono avviati un particolare insieme di demoni, che garantiscono un certo insieme di servizi.

3 Alcune caratteristiche specifiche di Linux

Una delle caratteristiche peculiari di Linux rispetto ad altri kernel Unix è quella di essere *modulare*; Linux cioè può essere esteso inserendo a sistema attivo degli ulteriori "pezzi", i *moduli*, che permettono di ampliare le capacità del sistema (ad esempio fargli riconoscere una nuova periferica).

Questi possono poi essere tolti dal sistema in maniera automatica quando non sono più necessari: un caso tipico è quello del modulo che permette di vedere il floppy, caricato solo quando c'è necessità di leggere un dischetto ed automaticamente rimosso una volta che non sia più in uso per un certo tempo.

In realtà è sempre possibile costruire Linux inserendo tutti i moduli che servono al suo interno, ottenendo quello che viene chiamato un kernel *monolitico* (come sono tutti i kernel degli altri unix); questo permette di evitare il ritardo nel caricamento dei moduli al momento della richiesta, ma comporta un maggiore consumo di memoria (dovendo tenere dentro anche codice non utilizzato), ed una minore flessibilità in quanto si perde la capacità di poter specificare eventuali opzioni al momento del caricamento.

Per contro l'uso dei moduli degrada leggermente le prestazioni e può dar luogo a conflitti inaspettati (che con un kernel monolitico avrebbero bloccato il sistema all'avvio), questi problemi oggi sono sempre più rari; in ogni caso non è possibile utilizzare i moduli nel caso in cui la funzionalità da essi fornite siano necessarie ad avviare il sistema.

Una seconda peculiarità di Linux è quella del Virtual File System (o VFS). Un concetto generale presente in tutti i sistemi Unix (e non solo) è che lo spazio su disco su cui vengono tenuti i file di dati è organizzato in quello che viene chiamato un *filesystem*. Lo spazio grezzo, che è normalmente diviso in settori contigui di dimensione fissa, viene cioè organizzato in maniera tale da permettere il rapido reperimento delle informazioni memorizzate su questi settori che possono essere sparsi sul disco, per presentarli in quello che l'utente vede come un file.

Quello che contraddistingue Linux è che l'interfaccia per la lettura del contenuto del filesystem è stata completamente modularizzata, per cui inserendo gli opportuni moduli nel sistema diventa possibile accedere con la stessa interfaccia (e, salvo limitazioni della realizzazione, in maniera completamente trasparente all'utente) ai più svariati tipi di filesystem, a partire da quelli usati da Windows e dal DOS, dal MacOS, e da tutte le altre versioni di Unix.

Dato che essa gioca un ruolo centrale nel sistema, torneremo in dettaglio sull'interfaccia dei file (e di come possa essere usata anche per altro che i file di dati) in 5; quello che è importante tenere presente da subito è che la disponibilità di una astrazione delle operazioni sui file rende Linux estremamente flessibile, dato che attraverso di essa è in grado di supportare con relativa

facilità, ed in maniera nativa, una varietà di filesystem superiore a quella di qualunque altro sistema operativo.

4 Un sistema multiutente

Linux, come ogni Unix, è nato come sistema multiutente e nella sua architettura è nativa la possibilità di avere utenti diversi che lavorano sulla stessa macchina. Inoltre l'architettura è strutturata in maniera tale che questa capacità non sia altro che una conseguenza della capacità del sistema di eseguire molti processi contemporaneamente.

Per evitare che utenti diversi possano danneggiarsi fra loro o interferire in maniera nociva, il sistema prevede un controllo degli accessi, che pone una serie di limitazioni a quello che un singolo utente può fare, associando ogni processo ad un utente, ed impedendo al processo l'esecuzione di tutte le operazioni non consentite.

Per questo motivo tutti gli Unix prevedono una procedura di autenticazione che permette di riconoscere l'utente che si collega al sistema. Questa nella sua forma più elementare è fatta dal programma `login`, che richiede all'utente il nome che lo identifica di fronte al sistema (detto *username*) ed una password che ne permette di verificare l'identità.

Gli utenti poi possono venire raggruppati in *gruppi*, ed anche ogni gruppo ha un nome (detto *group name*); inoltre ogni utente è sempre associato almeno un *gruppo* (detto *gruppo di default*, che di solito contiene solo l'utente in questione e ha nome uguale all'username). Un utente può comunque appartenere a più gruppi, che possono cosi venire usati per permettere l'accesso ad una serie di risorse comuni agli utenti dello stesso gruppo.

La gestione dei privilegi e degli utenti è tutto sommato piuttosto semplice, tanto da essere in certi casi considerata troppo primitiva; esistono estensioni che permettono di renderla più sottile e flessibile, ma nella maggior parte dei casi il controllo di accesso standard è più che sufficiente. In sostanza per il sistema esistono tre livelli di privilegi:

- i privilegi dell'utente (indicati con la lettera `u`, dall'inglese *user*).
- i privilegi del gruppo (indicati con la lettera `g`, dall'inglese*group*).
- i privilegi di tutti gli altri (indicati con la lettera `o`, dall'inglese *other*).

e tre permessi base:

- permesso di lettura (indicato con la lettera `r`, dall'inglese *read*).
- permesso di scrittura (indicato con la lettera `w`, dall'inglese *write*).
- permesso di esecuzione (indicato con la lettera `x`, dall'inglese *execute*).

il cui significato è abbastanza ovvio.

Ogni file è associato ad un utente, che è detto *proprietario* del file e ad un gruppo; per ciascuno dei tre livelli di privilegio (utente, gruppo e altri) è possibile specificare uno dei tre permessi; questi vengono riportati nella versione estesa dell'output del comando `ls`:

```
[piccardi@gont gapil]$ ls -l sources/
drwxr-sr-x    2 piccardi piccardi      128 Jan  4 00:46 CVS
-rw-r--r--    1 piccardi piccardi     3810 Sep 10 00:45 ElemDaytimeTCPClient.c
-rw-r--r--    1 piccardi piccardi     4553 Sep  9 19:39 ElemDaytimeTCPCuncServ.c
-rw-r--r--    1 piccardi piccardi     3848 Sep 10 00:45 ElemDaytimeTCPServer.c
-rw-r--r--    1 piccardi piccardi     3913 Sep 10 00:45 ElemEchoTCPClient.c
-rw-r--r--    1 piccardi piccardi     4419 Sep  9 19:39 ElemEchoTCPServer.c
```

```
-rw-r--r--    1 piccardi piccardi     9534 Oct 14 17:05 ErrCode.c
-rw-r--r--    1 piccardi piccardi     4103 Oct 26 23:40 ForkTest.c
-rw-r--r--    1 piccardi piccardi     1052 Jan  1 12:53 Makefile
-rw-r--r--    1 piccardi piccardi     3013 Jan  4 00:44 ProcInfo.c
-rw-r--r--    1 piccardi piccardi     3904 Sep 10 00:45 SimpleEchoTCPClient.c
-rw-r--r--    1 piccardi piccardi     4409 Sep 10 00:45 SimpleEchoTCPServer.c
-rw-r--r--    1 piccardi piccardi     1748 Sep 10 00:45 SockRead.c
-rw-r--r--    1 piccardi piccardi     1678 Sep 10 00:45 SockWrite.c
-rw-r--r--    1 piccardi piccardi     2821 Jan  4 00:44 TestRen.c
-rwxr-xr-x    1 piccardi piccardi    28944 Jan  1 13:11 getparam
-rw-r--r--    1 piccardi piccardi     4416 Jan  4 00:44 getparam.c
-rw-r--r--    1 piccardi piccardi     3018 Jan  4 00:44 test_fopen.c
-rw-r--r--    1 piccardi piccardi     7404 Jun 10  2001 wrappers.h
```

ci mostra nella prima colonna i permessi secondo lo schema riportato in fig. 2. La lettura dell'output di questo comando ci permette di vedere che i sorgenti dei programmi contenuti nella directory in oggetto hanno quelli che sono in genere i permessi di default per i file di dati, e cioè il permesso di scrittura solo per il proprietario, quello di lettura per tutti quanti (proprietario, gruppo e altri) e l'assenza del permesso di esecuzione.

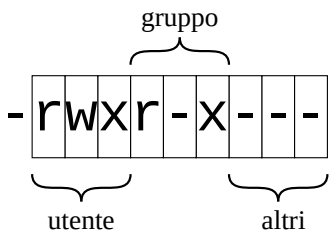

Figura 2: Legenda dei permessi dell'output di `ls`.

Nel caso di un programma invece (come `getparam` nell'esempio) i permessi di default sono diversi, ed il permesso di esecuzione è abilitato per tutti; lo stesso vale per le directory, dato che in questo caso il permesso di esecuzione ha il significato che è possibile attraversare la directory quando si scrive un pathname.

In generale un utente può effettuare su un file solo le azioni per le quali ha i permessi, questo comporta ad esempio che di default un utente può leggere i file di un altro ma non può modificarli o cancellarli; il proprietario di un file può sempre modificarne i permessi (con il comando `chmod`) per allargarli o restringerli (ad esempio i file della posta elettronica normalmente non hanno il permesso di lettura).

Tutto questo vale per tutti gli utenti con una eccezione, l'utente associato all'amministratore del sistema, che storicamente è chiamato `root`. Questo utente non è soggetto a nessun tipo di restrizione e può eseguire qualunque operazione; in genere è a questo utente che appartengono tutti i file di configurazione ed i programmi installati nel sistema, cosicché diventa impossibile per un utente normale poter danneggiare (accidentalmente o meno) i file del sistema[4].

5 L'organizzazione dei file

Come accennato in sez. 3 i file sono organizzati sui dischi all'interno di filesystem. Perché i file diventino accessibili al sistema un filesystem deve essere *montato*. Questa è una operazione pri-

[4]questa è una delle ragioni per cui i virus sono molto più difficili da fare con Linux: non essendo possibile ad un utente normale modificare i file di sistema, un virus potrà al più infettare i file di quell'utente, cosa che rende molto più difficile la sua diffusione

vilegiata (che normalmente può fare solo l'amministratore) che provvede ad installare nel kernel le opportune interfacce (in genere attraverso il caricamento dei relativi moduli) che permettono l'accesso ai file contenuti nel filesystem.

Come esempio consideriamo il caso in cui si voglia leggere il contenuto di un CD. Il kernel dovrà poter disporre sia delle interfacce per poter parlare al dispositivo fisico (ad esempio il layer della SCSI, se il CDROM è SCSI), che di quelle per la lettura dal dispositivo specifico (il modulo che si interfaccia ai CDROM, che è lo stesso che questi siano su SCSI, IDE o USB), sia di quelle che permettono di interpretare il filesystem ISO9660 (che è quello che viene usato per i dati registrati su un CDROM) per estrarne il contenuto dei file.

Allo stesso modo se si volessero leggere i dati su un dischetto occorrerebbe sia il supporto per l'accesso al floppy, che quello per poter leggere il filesystem che c'è sopra (ad esempio *vfat* per un dischetto Windows e *hfs* per un dischetto MacOs).

Un aspetto fondamentale nella architettura dei file, che è sempre bene tenere presente, è che uno dei criteri base del design dei sistemi Unix è quello espresso dalla frase *everything is a file*. Questo comporta una serie di differenze nella gestione dei file rispetto ad altri sistemi.

Anzitutto in Unix tutti i file di dati sono uguali (non esiste la differenza fra file di testo o binari che c'è in Windows, né fra file sequenziali e ad accesso diretto che c'era nel VMS). Inoltre le estensioni sono solo convenzioni, e non significano nulla per il kernel, che legge tutti i file alla stessa maniera.

Inoltre il sistema vede come file anche le periferiche, attraverso dei file speciali detti *device file* o *file di dispositivo*. Così si può suonare una canzone scrivendo su `/dev/dsp`, o leggere l'output di una seriale direttamente da `/dev/ttyS0`, o leggere direttamente dai settori fisici dell'hard-disk accedendo a `/dev/hda`, o fare animazioni scrivendo su `/dev/fb` (questo è molto più difficile da fare a mano).

Infine, in filesystem di tipo Unix, anche le directory non sono altro che una altro tipo di file speciale che contengono una lista di nomi associati ad un puntatore alla sezione del filesystem dove si trova l'informazione relativa al file in questione. Tutte le proprietà di un file (i permessi, la lunghezza, l'elenco delle zone del disco su cui sono immagazzinati i dati del contenuto) sono mantenuti in una apposita struttura chiamata *inode*. Per questo in realtà cancellare un file da una directory significa solo rimuovere da essa la voce con il puntatore all'inode.

Questo ci permette di capire anche un comportamento che può sembrare anomalo, quello per cui, in certe situazioni, pur cancellando un file non si recupera spazio disco. Il nome del file infatti non è una proprietà del file (e non sta nell'inode) ma solo un'etichetta in una directory. Uno stesso inode può essere perciò referenziato in più directory (ed un file quindi può avere quindi più nomi, anche completamente scorrelati fra loro), dando luogo a quello che si chiama un `hard link`.

Se torniamo all'output del comando `ls -l`, il numero presente nella seconda colonna indica proprio il numero di riferimenti presente per ciascun inode. Questo permette di ottenere un riferimento immediato allo stesso file (la cosa funziona solo all'interno di un filesystem, per poter fare un link a file su un altro filesystem occorre usare i *link simbolici*), ma fintanto che tutti i riferimenti ad un inode non sono stati cancellati il kernel non libererà lo spazio su disco occupato dal file.

6 La struttura delle directory

Una delle caratteristiche specifiche di un sistema Unix è che l'albero delle directory è unico; non esistono i vari dischi (o volumi) che si possono trovare in altri sistemi, come su Windows o su MacOS. All'avvio il kernel monta quella che si chiama la *directory radice* (o *root directory*) dell'albero (che viene indicata con `/`), tutti i restanti dischi, il CDROM, il floppy ed qualunque

altro dispositivo di memorizzazione dei dati, verranno poi montati successivamente in opportune sotto-directory della radice.

In maniera analoga a come lo si è montato, quando non si ha più la necessità di accedere ad un filesystem, questo potrà essere *smontato*. In questo modo diventa possibile rimuovere (nel caso di kernel modulare) le eventuali risorse aggiuntive, e liberare la directory utilizzata per il montaggio per il riutilizzo[5].

I nomi dei file sono indicati con un *pathname* o *percorso*, che descrive il cammino che occorre fare nell'albero per raggiungere il file passando attraverso le varie directory; i nomi delle directory sono separati da delle `/`. Il percorso può essere indicato (vedi tab. 1) in maniera assoluta, partendo dalla directory radice, o in maniera relativa, partendo dalla *directory corrente* (quest'ultima può essere settata dal comando `cd` e stampata con il comando `pwd`).

Esempio	Formato
`/home/piccardi/gapil/gapil.tex`	assoluto
`gapil/gapil.tex`	relativo

Tabella 1: Formato dei pathname

Come per il processo `init`, che non è figlio di nessun altro processo, anche la directory radice non è figlia di nessuna altra directory, come accennato in sez. 1 essa viene montata direttamente dal kernel in fase di avvio. Per questo motivo la directory radice viene ad assumere un ruolo particolare, ed il filesystem che la supporta deve contenere tutti i programmi di sistema necessari all'avvio (`init` compreso). Per questo `/` è l'unica directory che non può venire smontata, e può essere cambiata solo con un riavvio.

La organizzazione dell'albero delle directory è standardizzata in maniera molto accurata da un documento che si chiama *Filesystem Hierarchy Standard*, a cui tutte le distribuzioni si stanno adeguando. Lo standard descrive in dettaglio la struttura delle directory e del loro contenuto, prevedendo una divisione molto rigorosa che permette una notevole uniformità anche fra distribuzioni diverse, ed organizza in maniera meticolosa ed ordinata dati, programmi, file di configurazione, ecc.

Directory	Contenuto
`/bin`	comandi essenziali
`/boot`	file statici necessari al bootloader
`/dev`	file di dispositivo
`/etc`	file di configurazione della macchina
`/lib`	librerie essenziali e moduli del kernel
`/mnt`	mount point per filesystem temporanei
`/opt`	pacchetti software addizionali
`/sbin`	comandi di sistema essenziali
`/tmp`	file temporanei
`/usr`	gerarchia secondaria
`/var`	dati variabili

Tabella 2: Sottodirectory di `/` obbligatorie

In particolare le directory vengono suddivise sulla base di due criteri fondamentali; il primo è quello della possibilità di contenere file il cui contenuto può essere modificato (nel qual caso il filesystem che le contiene deve essere montato in lettura/scrittura) o meno (nel qual caso il

[5] con le ultime versioni di kernel in realtà questo non è più necessario, in quanto si possono *impilare* più montaggi sulla stessa directory; è comunque necessario smontare un device rimovibile come il floppy o il CD, prima di poterlo estrarre e sostituire.

filesystem può essere montato in sola lettura); il secondo è quello della possibilità di contenere file che possono essere condivisi (come i programmi di sistema) fra più stazioni di lavoro (ad esempio utilizzando un filesystem di rete) o file che invece sono locali e specifici alla macchina in questione.

Lo standard prevede che debbano essere necessariamente presenti le sottodirectory di `/` specificate in tab. 2, mentre quelle di tab. 3 sono obbligatorie soltanto qualora si siano installati i sottosistemi a cui essi fanno riferimento (utenti, `/proc` filesystem, diversi formati binari)[6].

Directory	Contenuto
`/lib<qual>`	librerie in formati alternativi
`/home`	home directory degli utenti
`/root`	home directory di *root*
`/proc`	filesystem virtuale con le informazioni sul sistema

Tabella 3: Sottodirectory di `/` obbligatorie in caso di utenti

Un elenco delle specifiche delle caratteristiche e del contenuto di ciascuna delle sottodirectory di `/` è riportato di seguito; per alcune di esse, come `/usr` e `/var`, sono previste delle ulteriori sottogerarchie che definiscono ulteriori dettagli dell'organizzazione dei file.

`/bin` Contiene i comandi essenziali del sistema (usati sia dall'amministratore che dagli utenti), che devono essere disponibili anche quando non ci sono altri filesystem montati (ad esempio quando si è un *single user mode*). Non deve avere sottodirectory e non può essere montata su un filesystem diverso da quello della radice.

`/boot` Contiene tutti i file necessari al procedimento di boot (immagini del kernel, ramdisk, ecc.) eccetto i file di configurazione ed i programmi per il settaggio del procedimento stesso (che vanno in `/sbin`). Può essere su qualunque filesystem purché visibile dal bootloader.

`/dev` Contiene i file di dispositivo, che permettono l'accesso alle periferiche. Non può essere montata su un filesystem diverso da quello di `/`.

`/etc` Contiene i file di configurazione del sistema e gli script di avvio. Non deve contenere programmi binari e non può essere montata su un filesystem diverso da quello di `/`. I file possono essere raggruppati a loro volta in directory; lo standard prevede solo che, qualora siano installati, siano presenti le directory `/etc/opt` (per i pacchetti opzionali), `/etc/X11` (per la configurazione di X Windows) e `/etc/sgml` (per la configurazione di SGML e XML).

`/home` Contiene le home directory degli utenti, la sola parte del filesystem (eccetto `/tmp`) su cui gli utenti hanno diritto di scrittura. Può essere montata su qualunque filesystem.

`/lib` Contiene le librerie condivise essenziali, usate dai programmi di `/bin` e `/sbin`, e deve essere sullo stesso filesystem di `/`. Qualora sia stato installato un kernel modulare i moduli devono essere installati in `/lib/modules`.

`/mnt` Contiene i mount point per i filesystem temporanei ad uso dell'amministratore di sistema (i filesystem di periferiche permanenti come i floppy o il CDROM possono essere tenuti sia in questa directory che direttamente sotto `/`).

[6] le `/lib` alternative sono state usate al tempo della transizione dal formato *a.out* ad *ELF*, oggi sono in completo disuso.

`/opt` Contiene eventuali pacchetti software aggiuntivi. Può essere su qualunque filesystem. Un pacchetto deve installarsi nella directory `/opt/package` dove *package* è il nome del pacchetto. All'amministratore è riservato l'uso delle directory opzionali `/opt/bin`, `/opt/doc`, `/opt/include`, `/opt/info`, `/opt/lib` e `/opt/man`. File variabili attinenti ai suddetti pacchetti devono essere installati in `/var/opt` e i file di configurazione in `/etc/opt`, nessun file attinente ai pacchetti deve essere installato al di fuori di queste directory.

`/root` È la home directory dell'amministratore.

`/sbin` Contiene i programmi essenziali per l'amministrazione del sistema (come `init`). Deve essere sullo stesso filesystem di `/`. Vanno messi in questa directory solo i programmi essenziali per il boot, il recupero e la manutenzione dei filesystem.

`/tmp` La directory viene usata per mantenere file temporanei. Viene cancellata ad ogni riavvio, ed i programmi non devono assumere che i file siano mantenuti fra due esecuzioni successive.

`/usr` È la directory principale che contiene tutti i file ed i dati non variabili che possono essere condivisi fra più stazioni. Di solito viene montata su un filesystem separato rispetto a `/` in modo da poterla montare in sola lettura. Prevede una ulteriore gerarchia di directory in cui i vari file vengono organizzati; lo standard richiede obbligatoriamente seguenti:

- `bin` Contiene i programmi usati dall'utente installati dal sistema (o dalla distribuzione originale). Non può essere ulteriormente suddivisa.
- `include` Contiene tutti gli header file usati dal compilatore e dai programmi C.
- `lib` Contiene le librerie relative ai programmi di `bin` e `sbin`.
- `local` Contiene una replica della gerarchia di `/usr` dedicata ai file installati localmente dall'amministratore. In genere qui vengono installati i programmi compilati dai sorgenti originali e tutto quello che non fa parte della distribuzione ufficiale.
- `sbin` Contiene le utilità di sistema non essenziali, ad uso dell'amministratore.
- `share` Contiene una gerarchia in cui sono organizzati tutti i dati che non dipendono dalla architettura hardware: `man` per le pagine di manuale, `dict` per i dizionari, `doc` per la documentazione, `games` per i dati statici dei giochi, `info` per i file del relativo sistema di help, `terminfo` per il database con le informazioni sui terminali, `misc` per tutto quello che non viene classificato nelle altre.

mentre sono obbligatorie solo se i relativi pacchetti sono installati, le seguenti directory:

- `X11R6` Contiene la gerarchia dei file relativi ad X Window.
- `games` Contiene i binari dei giochi.
- `src` Contiene i sorgenti dei pacchetti.

`/var` Contiene i file variabili: le directory di spool, i file di log e amministrativi, dati transienti e temporanei, in modo che `/usr` possa essere montata in sola lettura. È preferibile montarla in un filesystem separato; alcune directory non possono essere condivise. Anche in questo caso i file sono organizzati in una gerarchia standardizzata che prevede le seguenti sottodirectory:

- `cache` Dati di appoggio per le applicazioni.
- `lib` Informazioni variabili sullo stato del sistema
- `local` Dati variabili relativi ai pacchetti di `/usr/local`.

`lock` File di lock.

`opt` File variabili per i pacchetti di `/opt`.

`run` Dati relativi ai processi in esecuzione.

`spool` Directory per i dati di spool di varie applicazioni (stampanti, posta elettronica, news, ecc.).

`tmp` File temporanei non cancellati al riavvio del sistema.

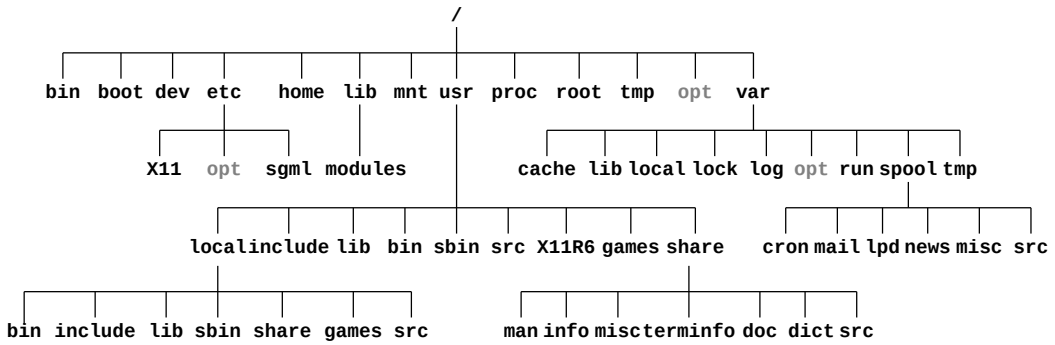

Figura 3: Struttura tipica delle directory, secondo il Filesystem Hierarchy Standard.

In fig. 3 è riportata una rappresentazione grafica della struttura generale delle directory prevista dal FHS, (si è mostrata solo una parte delle directory previste). I dettagli completi sulla struttura (così come le specifiche relative ad i contenuti delle varie directory, possono essere reperiti sul documento ufficiale di definizione del FHS, disponibile all'indirizzo: http://www.pathname.com/fhs/.