Amministrare GNU/Linux

Simone Piccardi

23 marzo 2004

Indice

Capitolo 1

L'architettura di un sistema GNU/Linux

1.1 L'architettura del sistema.

Prima di addentrarci nei dettagli di funzionamento di un sistema GNU/Linux, conviene fornire un quadro generale per introdurre i vari concetti su cui si basa l'architettura di questo sistema, che è basata su quella, consolidatasi in 30 anni di impiego, dei sistemi di tipo Unix.

Il fatto che questa architettura abbia una certa età fa si che spesso i detrattori di GNU/Linux ne denuncino la presunta mancanza di *innovatività*, ma anche le case hanno da secoli le stesse basi architetturali (porte, muri e tetti), ma non per questo non esiste innovazione. Il vantaggio della architettura di Unix infatti è quello di aver fornito una solida base per la costruzione di sistemi affidabili ed efficienti, consolidata e corretta in decenni di utilizzo, tanto che ormai è divenuto comune il detto che *chi non usa l'architettura Unix è destinato a reinventarla*.

1.1.1 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. Questo 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 base dell'architettura di ogni sistema Unix come GNU/Linux è quello di una rigida separazione fra il *kernel* (il *nucleo* del sistema, cui si demanda la gestione delle risorse hardware, come la CPU, la memoria, le periferiche) e i *processi*, (le unità di esecuzione dei programmi, che nel caso vanno dai comandi base di sistema, agli applicativi, alle interfacce per l'interazione con gli utenti).

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, viene realizzato usando processi che eseguono gli opportuni programmi.

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. 1.1.

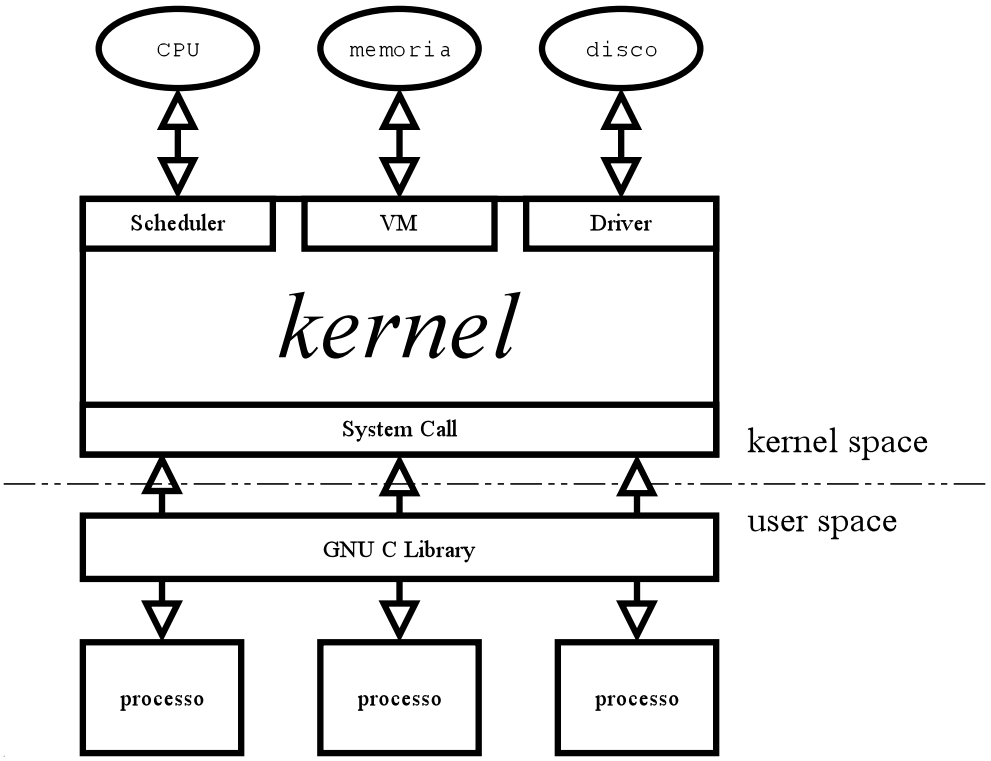

Figura 1.1: Struttura del sistema

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 vedono se stessi 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 nelle impostazioni del BIOS un apposito programma, il *bootloader*,[1] che a sua volta recupera (in genere dal disco) una immagine del kernel che viene caricata in memoria ed eseguita.

Una volta che il controllo è passato al kernel questo, terminata la fase di inizializzazione (in cui ad esempio si esegue una scansione delle periferiche disponibili, e si leggono le tabelle delle partizioni dei vari dischi) si incaricherà di montare (vedi sez. 1.2.2) il filesystem su cui è situata la *directory radice* (vedi sez. 1.2.3), e farà partire il primo processo. Per convenzione questo processo si chiama `init`, ed è il programma di inizializzazione che a sua volta si cura di far partire tutti gli altri processi che permettono di usare il sistema.

[1] questo è un programma speciale, il cui solo compito è quello di far partire un sistema operativo, in genere ogni sistema ha il suo, nel caso di Linux per l'architettura PC i due principali sono LILO e GRUB, che vedremo in sez. 5.3.

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 (su cui torneremo in cap. 2) 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 quelli per entrare nel sistema, siano dei normali programmi come tutti gli altri, che vengono eseguiti dal kernel e fanno operazioni attraverso le funzioni (dette *system call*) che esso mette a disposizione, esattamente come accadrebbe se si fosse eseguito 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)[2] che altri sistemi operativi (come Windows) hanno al loro interno: tutte le operazioni di normale amministrazione di un sistema sono sempre realizzate tramite dei normali programmi.

1.1.2 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[3]), 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, il kernel, che benché fondamentale non costituisce da sola un sistema operativo.

Anche se il kernel tratta tutti i programmi allo modo, non tutti hanno la stessa importanza. 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. Una caratteristica fondamentale dell'architettura Unix infatti (ci torneremo in sez. 1.3) è quella per cui qualunque processo può a sua volta avviarne di nuovi,[4] per cui sarà cura di `init` mettere in esecuzione tutti i programmi necessari al funzionamento del sistema.

Benché in teoria sia possibile far partire qualunque altro programma al posto di `init`[5] tutti i sistemi Unix usano questo specifico programma come primo processo lanciato all'avvio del sistema. 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

[2] questa infatti viene eseguita usando semplicemente le funzioni che permettono di leggere e scrivere il contenuto di un file, leggendo l'originale e scrivendo sulla copia.

[3] in rigoroso ordine alfabetico!

[4] nel qual caso si dice che il primo processo è il *padre* degli altri, che a loro volta sono chiamati *figli*.

[5] 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.

grafica, e a seconda di quanto deciso dall'amministratore[6] si avrà un server di posta, o un server web, ecc.

1.1.3 Alcune caratteristiche specifiche di Linux

Benché Linux stia diventando il più diffuso, esistono parecchi altri kernel unix-like, sia liberi che proprietari, nati nella tumultuosa e complessa evoluzione che dallo Unix originario della AT/T ha portato alla nascita di una miriade di sistemi derivati (BSD, Solaris, AIX, HP-UX, Digital Unix, IRIX, solo per citare i più noti) che si innestano tutti in due rami principali, quelli derivati dal sistema sviluppato dalla AT/T, detto SysV (da *System V* ultima versione ufficiale) e quelli derivati dal codice sviluppato all'università di Berkley, detto BSD (da *Berkley Software Distribution*). La prima caratteristica distintiva di Linux è che esso è stato riscritto da zero, per cui non è classificabile in nessuno di questi due rami e prende invece, a seconda dei casi, le migliori caratteristiche di ciascuno di essi.

Un'altra delle caratteristiche peculiari di Linux rispetto agli altri kernel unix-like 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 un kernel Linux comprensivo di tutti i moduli che servono, ottenendo quello che viene chiamato un kernel *monolitico* (come sono 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 il kernel anche codice non utilizzato), ed una flessibilità nettamente inferiore in quanto si perde la capacità di poter specificare eventuali opzioni al momento del caricamento, costringendo al riavvio in caso di necessità di cambiamenti.

Per contro in certi casi l'uso dei moduli può degradare leggermente (quasi sempre in maniera assolutamente non avvertibile) 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 virtualizzata, 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 sez. 1.2; 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

[6] di norma lo si fa in fase di installazione, ma lo si può cambiare anche in seguito.

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

1.2 L'architettura dei file

Un aspetto fondamentale della architettura di GNU/Linux è quello della gestione dei file, esso deriva direttamente da uno dei criteri base della progettazione di tutti i sistemi Unix, quello espresso dalla frase *everything is a file* (cioè *tutto è un file*), per cui l'accesso ai file e alle periferiche è gestito attraverso una interfaccia identica.

Inoltre, essendo in presenza di un sistema multiutente e multitasking, il kernel deve anche essere in grado di gestire l'accesso contemporaneo allo stesso file da parte di più processi, e questo viene fatto usando un design specifico nella struttura delle interfacce di accesso, che è uno dei punti di maggior forza della architettura di un sistema Unix.

1.2.1 Il *Virtual File System* e le caratteristiche dei file.

Come accennato in sez. 1.1.3 i file sono organizzati sui dischi all'interno di filesystem. Perché i file diventino accessibili al sistema un filesystem deve essere *montato* (torneremo su questo in sez. 1.2.5). Questa è una operazione privilegiata (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 di solito 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).

Come accennato nell'introduzione a questa sezione, uno dei criteri fondamentali dell'architettura di un sistema Unix è quello per cui *tutto è un file* e che altro non significa che si può accedere a tutte le periferiche[7] con una interfaccia identica a quella con cui si accede al contenuto dei file. Questo comporta una serie di differenze nella gestione dei file rispetto ad altri sistemi.

Anzitutto in un sistema 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 di dati alla stessa maniera, indipendentemente dal nome e dal contenuto.

In realtà il sistema prevede tipi diversi di file, ma in un altro senso; ad esempio il sistema può accedere alle periferiche, attraverso dei file speciali detti *device file* o *file di dispositivo*. Così si può suonare una canzone scrivendo su `/dev/dsp`, leggere l'output di una seriale direttamente da `/dev/ttyS0`, leggere direttamente dai settori fisici dell'harddisk accedendo a `/dev/hda`, o fare animazioni scrivendo su `/dev/fb0` (questo è molto più difficile da fare a mano). Un elenco dei vari tipi oggetti visti come file dal kernel è riportato in tab. 1.1, ognuno di questi fornisce una una funzionalità specifica, sempre descritta in tabella.

Altri tipi di file speciali sono le *fifo* ed i *socket*, che altro non sono che dei canali di comunicazione messi a disposizione dei processi perché questi possano comunicare fra loro. Dato che i processi sono completamente separati deve essere il kernel a fornire le funzionalità che permettano la comunicazione. Questi file speciali sono due modalità per realizzare questa comunicazione.

[7] con la sola eccezione delle interfacce ai dispositivi di rete, che non rientrano bene nell'astrazione.

Tipo di file			Descrizione
regular file	*file regolare*	`-`	un file che contiene dei dati (l'accezione normale di file)
directory	*cartella o direttorio*	`d`	un file che contiene una lista di nomi associati a degli *inode*.
symbolic link	*collegamento simbolico*	`l`	un file che contiene un riferimento ad un altro file/directory
char device	*dispositivo a caratteri*	`c`	un file che identifica una periferica ad accesso a caratteri
block device	*dispositivo a blocchi*	`b`	un file che identifica una periferica ad accesso a blocchi
fifo	"*coda*"	`f`	un file speciale che identifica una linea di comunicazione unidirezionale.
socket	"*presa*"	`s`	un file speciale che identifica una linea di comunicazione bidirezionale.

Tabella 1.1: I vari tipi di file riconosciuti da Linux

Aprendo una *fifo* un processo può scrivervi sopra ed un altro processo leggerà dall'altro capo quanto il primo ha scritto, niente verrà salvato su disco, ma passerà tutto attraverso il kernel che consente questa comunicazione come attraverso un tubo. I *socket* fanno la stessa cosa ma consentono una comunicazione bidirezionale, in cui il secondo processo può scrivere indietro, ed il primo leggere, quando invece per le *fifo* il flusso dei dati è unidirezionale.

La possibilità di avere tutti questi tipi di file speciali avviene anche grazie al fatto che in Linux l'accesso ai file viene effettuato attraverso una interfaccia, detta *Virtual File System*, che prevede una serie di operazioni generiche applicabili ad un qualunque oggetto del sistema il quale, secondo la filosofia del *tutto è un file*, è accessibile tramite essa.

Le principali operazioni sono riportate in tab. 1.2; ogni oggetto del sistema visto attraverso il *Virtual File System* definisce la sua versione di queste operazioni. Come si può notare sono definite sia operazioni generiche come la lettura e la scrittura, che più specialistiche come lo spostamento all'interno di un file.[8] Quando si utilizzano le system call per accedere ad un file sarà compito del kernel chiamare l'operazione relativa ad esso associata (che sarà ovviamente diversa a seconda del tipo di file), o riportare un errore quando quest'ultima non sia definita (ad esempio sul file di dispositivo associato alla seriale non si potrà mai definire l'operazione di spostamento `llseek`).

Funzione	Operazione
`open`	apre il file.
`read`	legge dal file.
`write`	scrive sul file.
`llseek`	si sposta all'interno del file.
`ioctl`	accede alle operazioni di controllo.
`readdir`	legge il contenuto di una directory.

Tabella 1.2: Principali operazioni sui file definite nel VFS.

Il *Virtual File System* è anche il meccanismo che permette al kernel di gestire tanti filesystem diversi; quando uno di questi viene montato è compito il kernel utilizzare per le varie system call le opportune operazioni in grado di accedere al contenuto di quel particolare filesystem; questa è la ragione principale della grande flessibilità di Linux nel supportare i filesystem più diversi, basta definire queste operazioni per un filesystem per poterne permettere l'accesso da parte delle varie system call secondo la stessa interfaccia.

Uno dei comandi fondamentali per la gestione dei file è `ls` (il cui nome deriva da *LiSt file*),

[8] ed altre ancora più complesse non sono state riportate.

il comando mostra l'elenco dei file nella directory corrente. Usando l'opzione `-l` è possibile ottenere una lista estesa, in cui compaiono varie proprietà del file; ad esempio:

```
piccardi@oppish:~/filetypes$ ls -l
total 1
brw-r--r--    1 root     root       1,   2 Jul  8 14:48 block
crw-r--r--    1 root     root       1,   2 Jul  8 14:48 char
drwxr-xr-x    2 piccardi piccardi       48 Jul  8 14:24 dir
prw-r--r--    1 piccardi piccardi        0 Jul  8 14:24 fifo
-rw-r--r--    1 piccardi piccardi        0 Jul  8 14:24 file
lrwxrwxrwx    1 piccardi piccardi        4 Jul  8 14:25 link -> file
```

ci mostra il contenuto di una directory dove si sono creati i vari tipi di file[9] elencati in tab. 1.1, si noti come la prima lettera in ciascuna riga indichi il tipo di file, anche questo secondo la notazione riportata nella terza colonna della stessa tabella.

Il comando `ls` è dotato di innumerevoli opzioni, che gli consentono di visualizzare le varie caratteristiche dei file, delle quali il tipo è solo una. Altre caratteristiche sono i tempi di ultimo accesso, modifica e cambiamento (su cui torneremo fra poco), le informazioni relative a permessi di accesso e proprietari del file (che vedremo in dettaglio in sez. 1.4.2), la dimensione, il numero di hard link (che vedremo in sez. 1.2.2).

Il comando prende come parametro una lista di file o directory, senza opzioni viene mostrato solo il nome del file (se esiste) o il contenuto della directory specificata. Le opzioni sono moltissime, e le principali sono riportate in tab. 1.3; l'elenco completo è riportato nella pagina di manuale accessibile con il comando `man ls`.

Opzione	Significato
`-l`	scrive la lista in formato esteso.
`-a`	mostra i file *invisibili*.
`-i`	scrive il numero di `inode` (vedi sez. 1.2.2).
`-R`	esegue la lista ricorsivamente per tutte le sottodirectory.
`-c`	usa il tempo di ultimo cambiamento del file.
`-u`	usa il tempo di ultimo accesso al file.
`-d`	mostra solo il nome e non il contenuto quando riferito ad una directory, e non segue i link simbolici.

Tabella 1.3: Principali opzioni del comando `ls`.

Una convenzione vuole che i file il cui nome inizia per un punto (`.`) non vengano riportati nell'output di `ls`, a meno di non specificarlo esplicitamente con l'uso dell'opzione `-a`; per questo tali file sono detti *invisibili*. Si tenga presente comunque che questa *non* è una proprietà dei file e non ha nulla a che fare con le modalità con cui il kernel li tratta (che sono sempre le stesse), ma solo una convenzione usata e rispettata dai vari programmi in user space.

L'opzione `-l` permette di mostrare una lista in formato esteso in cui vengono riportate molte informazioni concernenti il file. Abbiamo visto in precedenza un esempio di questa lista, e come il primo carattere della prima colonna indichi tipo di file, il resto della colonna indica i *permessi* del file, secondo una notazione su cui torneremo in sez. 1.4.2. Il secondo campo indica il numero di *hard link* al file (su questo torneremo in sez. 1.2.2), mentre il terzo ed il quarto campo indicano rispettivamente utente e gruppo proprietari del file (anche questo sarà trattato in sez. 1.4.2). Il quinto campo indica la dimensione, usualmente riportata in byte.

[9] con l'eccezione dei *socket*, questi ultimi infatti non sono di norma utilizzati dai comandi di shell, ma vengono creati direttamente dai programmi che li usano (il caso più comune è X window), non esiste pertanto un comando che permetta di crearne uno in maniera esplicita.

Il sesto campo è il tempo di *ultima modifica* del file e l'ultimo campo il nome del file. I tempi dei file (mantenuti automaticamente dal kernel quando opera si essi) in un sistema unix-like sono tre ed hanno un significato diverso rispetto a quanto si trova in altri sistemi operativi. Il tempo mostrato di default da `ls` è il *tempo di ultima modifica* (o *modification time*) che corrisponde all'ultima volta che è stato modificato il contenuto di un file. Si badi bene che questo tempo riguarda solo il *contenuto* del file, se invece si operano delle modifiche sulle proprietà del file (ad esempio si cambiano i permessi) varia quello che viene chiamato *tempo di ultimo cambiamento* (il *change time*) che viene visualizzato con l'opzione `-c`. Infine tutte le volte che si accede al contenuto del file viene cambiato il *tempo di ultimo accesso* (o *access time*) che può essere visualizzato con l'opzione `-u`. Si noti infine come in un sistema unix-like *non* esista un tempo di creazione del file.

1.2.2 L'architettura di un filesystem e le proprietà dei file

Come già accennato Linux (ed ogni sistema unix-like) organizza i dati che tiene su disco attraverso l'uso di un filesystem. Una delle caratteristiche di Linux rispetto agli altri Unix è quella di poter supportare, grazie al VFS, una enorme quantità di filesystem diversi, ognuno dei quali ha una sua particolare struttura e funzionalità proprie. Per questo non entreremo nei dettagli di un filesystem specifico, ma daremo una descrizione a grandi linee che si adatta alle caratteristiche comuni di qualunque filesystem di sistema unix-like.

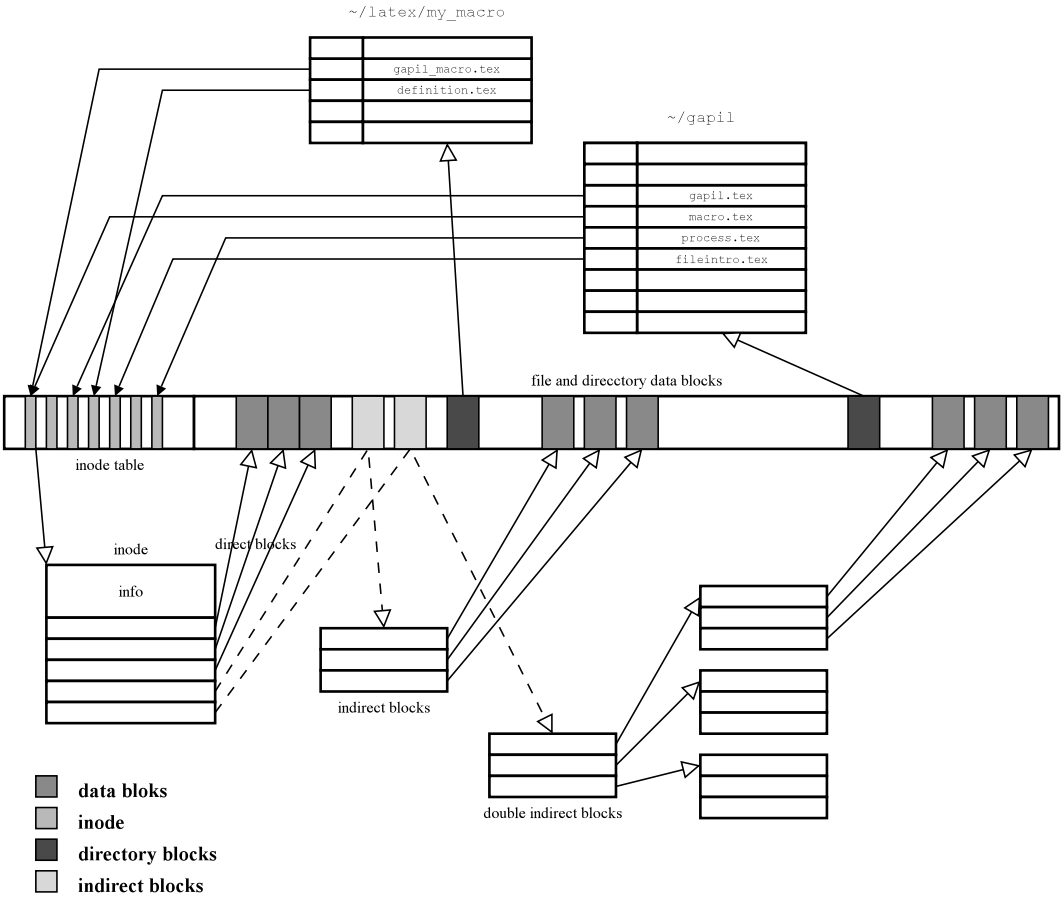

Figura 1.2: Strutturazione dei dati all'interno di un filesystem.

Se si va ad esaminare con maggiore dettaglio la strutturazione dell'informazione all'interno del singolo filesystem possiamo esemplificare la situazione con uno schema come quello esposto in fig. 1.2, da cui si evidenziano alcune delle caratteristiche di base di un filesystem, sulle quali è

bene porre attenzione visto che sono fondamentali per capire il funzionamento dei comandi che manipolano i file e le directory.

La struttura che identifica un file all'interno di un filesystem è il cosiddetto *inode*, a ciascun file infatti corrisponde un *inode*, che lo identifica univocamente. L'*inode* contiene tutte le informazioni riguardanti il file: il tipo di file, i permessi di accesso, le dimensioni, i puntatori ai blocchi fisici che contengono i dati e così via; le informazioni che il comando `ls` fornisce provengono dall'*inode*.

L'unica informazione relativa al file non contenuta nell'*inode* è il suo nome; infatti il nome di un file non è una proprietà del file, ma semplicemente una etichetta associata ad un *inode*. Le directory infatti non *contengono* i file, ma sono dei file speciali (di tipo *directory*, così che il kernel può trattarle in maniera diversa) il cui contenuto è semplicemente una lista di nomi a ciascuno dei quali viene associato un numero di *inode* che identifica il file cui il nome fa riferimento.

Come mostrato in fig. 1.2 si possono avere più voci in directory diverse che puntano allo stesso *inode*. Questo introduce il concetto di *hard link*: due file che puntano allo stesso *inode* sono fisicamente lo stesso file, nessuna proprietà specifica, come permessi, tempi di accesso o contenuto permette di distinguerli, in quanto l'accesso avviene per entrambi attraverso lo stesso *inode*.

Siccome uno stesso *inode* può essere referenziato in più directory, un file può avere più nomi, anche completamente scorrelati fra loro. Per questo ogni *inode* mantiene un contatore che indica il numero di riferimenti (detto *link count*) che gli sono stati fatti; questo viene mostrato nell'esempio di output di `ls` visto in precedenza, dal valore numerico riportato nel secondo campo, e ci permette di dire se un file ha degli *hard link* (anche se non possiamo sapere dove sono).

Il comando generico che permette di creare dei link è `ln` che prende come parametri il file originale ed il nome del link. La differenza rispetto a Windows è che in un sistema unix-like i link sono di due tipi, oltre agli hard link appena illustrati infatti esistono anche i cosiddetti *link simbolici*, o *symbolic link* (quelli più simili ai collegamenti di Windows o agli alias del MacOS), come il file `link` mostrato in precedenza.

Per creare un *hard link* basta usare direttamente il comando `ln` (da *LiNk file*), che di default crea questo tipo di link. Così potremo creare il file `hardlink` come *hard link* al file `file` visto in precedenza con il comando:

```
piccardi@oppish:~/filetypes$ ln file hardlink
```

e adesso potremo verificare che:

```
piccardi@oppish:~/filetypes$ ls -l
total 1
brw-r--r--    1 root     root       1,   2 Jul  8 14:48 block
crw-r--r--    1 root     root       1,   2 Jul  8 14:48 char
drwxr-xr-x    2 piccardi piccardi       48 Jul  8 14:24 dir
prw-r--r--    1 piccardi piccardi        0 Jul  8 14:24 fifo
-rw-r--r--    2 piccardi piccardi        0 Jul  8 14:24 file
-rw-r--r--    2 piccardi piccardi        0 Jul  8 14:24 hardlink
lrwxrwxrwx    1 piccardi piccardi        4 Jul  8 14:25 link -> file
```

e si noti come adesso il secondo campo mostri per `file` e `hardlink` un valore pari a due. Usando l'opzione `-i` di `ls` possiamo anche stampare per ciascun file il numero di *inode* ottenendo:

```
piccardi@oppish:~/filetypes$ ls -li
total 1
   2118 brw-r--r--    1 root     root       1,   2 Jul  8 14:48 block
```

```
2120 crw-r--r--    1 root     root       1,   2 Jul  8 14:48 char
  15 drwxr-xr-x    2 piccardi piccardi       48 Jul  8 14:24 dir
2115 prw-r--r--    1 piccardi piccardi        0 Jul  8 14:24 fifo
2117 -rw-r--r--    2 piccardi piccardi        0 Jul  8 14:24 file
2117 -rw-r--r--    2 piccardi piccardi        0 Jul  8 14:24 hardlink
2116 lrwxrwxrwx    1 piccardi piccardi        4 Jul  8 14:25 link -> file
```

e come si può notare `file` e `hardlink` hanno lo stesso numero di *inode* pari a 2117.

Il problema con gli *hard link* è che le directory contengono semplicemente il numero di *inode*, per cui si può fare riferimento solo ad un *inode* nello stesso filesystem della directory, dato che su un altro filesystem lo stesso numero identificherà un *inode* diverso. Questo limita l'uso degli *hard link* solo a file residenti sul filesystem corrente, ed il comando `ln` segnalerà un errore se si cerca di creare un *hard link* ad un file posto in un altro filesystem.

Per superare questa limitazione sono stati introdotti i link simbolici, che vengono creati usando l'opzione `-s` del comando `ln`; ad esempio si è creato il link simbolico `link` dell'esempio precedente con il comando `ln -s file link`. In questo caso viene creato un nuovo file, di tipo *symbolic link*, e con un suo diverso *inode*, come mostrato nell'esempio precedente, il cui contenuto sarà il nome del file a cui esso fa riferimento, che a questo punto può essere in qualsiasi altro filesystem. È compito del kernel far sì che quando si usa un link simbolico si vada poi ad usare il file a cui questo punta.

Oltre a `-s` il comando `ln` prende una serie di altre opzioni le principali delle quali sono riportate in tab. 1.4. La lista completa è riportata nella pagina di manuale accessibile attraverso il comando `man ln`.

Opzione	Significato
`-s`	crea un link simbolico.
`-f`	forza la sovrascrittura del nuovo file se esso esiste già.
`-i`	richiede conferma in caso di sovrascrittura.
`-d`	crea un *hard link* ad una directory (in Linux questa non è usabile).

Tabella 1.4: Principali opzioni del comando `ln`.

Una seconda caratteristica dei link simbolici è la possibilità di creare dei link anche per delle directory. Questa capacità infatti, sebbene teoricamente possibile anche per gli hard link, in Linux non è supportata per la sua pericolosità, è possibile infatti creare dei *link loop* se si commette l'errore di creare un link alla directory che contiene il link stesso; con un link simbolico questo errore può essere corretto in quanto la cancellazione del link simbolico rimuove quest'ultimo, e non il file referenziato, ma con un hard link non è più possibile fare questa distinzione e la rimozione diventa impossibile.

La possibilità di creare dei link alle directory tuttavia è estremamente utile, infatti qualora si voglia accedere ad una directory attraverso un path diverso (ad esempio a causa di un programma che cerca dei file di configurazione in una locazione diversa da quella usuale) piuttosto che dover spostare tutti i file basta creare un link simbolico e si sarà risolto il problema.

Oltre agli *hard link* la struttura di un filesystem unix-like ha ulteriori conseguenze non immediate da capire per chi proviene da sistemi operativi diversi. La presenza degli hard link e l'uso degli *inode* nelle directory infatti comporta anche una modalità diversa nella cancellazione dei file e nello spostamento degli stessi.

Il comando per la cancellazione di un file è `rm` (da *ReMove file*), ma la funzione usata dal sistema per effettuare questo compito si chiama in realtà `unlink` ed essa, come ci dice il nome, non cancella affatto i dati del file, ma si limita ad eliminare la relativa voce da una directory e decrementare il numero di riferimenti presenti nell'*inode*. Solo quando il numero di riferimenti ad un *inode* si annulla, i dati del file vengono effettivamente rimossi dal disco dal kernel. In realtà

oltre ai riferimenti mostrati da `ls` il kernel mantiene una ulteriore lista di riferimenti relativi a tutti i file che sono aperti, per cui anche se si cancellano tutti i nomi dalle varie directory in cui possono comparire, ma resta qualche processo attivo che ha aperto quel file, lo spazio disco non sarà rilasciato. 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 comando `rm` e prende come parametri una lista di file da cancellare; se si usa l'opzione `-i` il comando chiede di confermare la cancellazione, mentre con l'opzione `-f` si annulla ogni precedente `-i` ed inoltre non vengono stampati errori per file non esistenti. Infine l'opzione `-R` (o `-r`) permette la cancellazione ricorsiva di una directory e di tutto il suo contenuto, ed è pertanto da usare con estrema attenzione, specie se abbinata con `-f`. La lista completa delle opzioni è riportata nella pagina di manuale, accessibile con il comando `man rm`.

Come accennato la struttura di un filesystem unix-like comporta anche una diversa concezione dell'operazione di spostamento dei file, che nel caso è identica a quella di cambiamento del nome. Il comando per compiere questa operazione infatti è unico e si chiama `mv`, da *MoVe file*. Infatti fintanto che si "*sposta*" un file da una directory ad un altra senza cambiare filesystem, non c'è nessuna necessità di spostare il contenuto del file e basta semplicemente che sia creata una nuova voce per l'*inode* in questione rimuovendo al contempo la vecchia: esattamente la stessa cosa che avviene quando gli si cambia nome (nel qual caso l'operazione viene effettuata all'interno della stessa directory). Qualora invece si debba effettuare lo spostamento ad un filesystem diverso diventa necessario prima copiare il contenuto e poi cancellare l'originale.

Il comando `mv` ha due forme, e può prendere come argomenti o due nomi di file o una lista di file seguita da una directory. Nel primo caso rinomina il primo file nel secondo (cancellando quest'ultimo qualora esista già), nel secondo caso sposta tutti i file della lista nella directory (sovrascrivendo eventuali file presenti con lo stesso nome). Le principali opzioni sono riportate in tab. 1.5, l'elenco completo è riportato nella pagina di manuale, accessibile con il comando `man mv`.

Opzione	Significato
`-f`	forza la sovrascrittura del nuovo file se esso esiste già.
`-i`	richiede conferma in caso di sovrascrittura.
`-u`	esegue lo spostamento solo se la destinazione è più vecchia della sorgente.

Tabella 1.5: Principali opzioni del comando `mv`.

Dato che il comando si limita a cambiare di una voce associata ad un numero di *inode* all'interno di una directory, i tempi dei file non vengono mai modificati con l'uso di `mv`, fintanto che lo spostamento avviene all'interno dello stesso filesystem. Quando però lo spostamento avviene fra filesystem diversi viene copiato il contenuto e cancellato il file originario, pertanto in teoria dovrebbero risultare modificati anche i tempi di ultimo accesso e modifica. In realtà il comando provvede ripristinare questi tempi (come le altre caratteristiche del file) al valore del file originario, ma non può fare nulla per ripristinare il tempo di ultimo cambiamento.[10] Pertanto in quel caso si potrà notare, usando `ls -lc`, che questo è cambiato e corrisponde al momento dello spostamento.

Qualora invece si voglia duplicare un file il comando da usare è `cp` (da *CoPy file*). Come per `mv` può prendere come argomenti o due nomi di file o una lista di file seguita da una directory; nel primo caso effettua una copia del primo file sul secondo, nel secondo copia tutti file della lista nella directory specificata.

[10] il kernel fornisce delle system call che permettono di cambiare i tempi di ultimo accesso e modifica di un file, così che il comando `mv` può ripristinare i tempi precedenti, ma non ne esistono per cambiare il tempo di ultimo cambiamento, che corrisponderà pertanto al momento in cui il nuovo file è stato creato. Questa è una misura di sicurezza che permette sempre di verificare se un file è stato modificato, anche se si cerca di nascondere le modifiche.

Dato che il comando funziona copiando il contenuto di un file su un secondo file creato per l'occasione, i tempi di ultima modifica, accesso e cambiamento di quest'ultimo corrisponderanno al momento in cui si è eseguita l'operazione. Inoltre il file sarà creato con i permessi standard dell'utente che ha lanciato il comando, che risulterà anche il suo proprietario. Se si vogliono preservare invece le caratteristiche del file originale occorrerà usare l'opzione `-p`.

Opzione	Significato
`-f`	forza la sovrascrittura della destinazione se essa esiste già.
`-i`	richiede conferma in caso di sovrascrittura.
`-p`	preserva tempi, permessi e proprietari del file.
`-l`	crea degli hard link al posto delle copie.
`-s`	crea dei link simbolici al posto delle copie.
`-d`	copia il link simbolico invece del file da esso indicato.
`-r`	copia ricorsivamente tutto il contenuto di una directory.
`-R`	identico a `-R`.
`-a`	combina le opzioni `-dpR`.
`-L`	segue sempre i link simbolici.

Tabella 1.6: Principali opzioni del comando `cp`.

Si tenga presente poi che nel caso di link simbolici il comando copia il file indicato tramite il link, se invece si vuole vuole copiare il link stesso occorrerà usare l'opzione `-d`. Il comando permette inoltre di creare degli hard link invece che delle copie usando l'opzione `-l` e dei link simbolici usando l'opzione `-s`. Una lista delle principali opzioni è riportata in tab. 1.6, l'elenco completo è riportato nella pagina di manuale, accessibile attraverso il comando `man cp`.

1.2.3 La struttura dell'organizzazione delle directory

Un'altra delle caratteristiche specifiche di un sistema unix-like è che l'albero delle directory è unico; non esistono cioè i vari dischi (o volumi) che si possono trovare in altri sistemi, come su Windows, sul MacOS o sul VMS. All'avvio il kernel *monta*[11] 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* (vedi sez. 1.2.5) successivamente in opportune sotto-directory della radice.

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. 1.7) in maniera assoluta, partendo dalla directory radice, o in maniera relativa, partendo dalla cosiddetta *directory di lavoro corrente*.

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

Tabella 1.7: Formato dei pathname assoluti e relativi.

Quest'ultima è una caratteristica specifica di ogni processo, che viene ereditata dal padre alla sua creazione (vedi sez. 1.3.1). Quando si entra nel sistema la directory di lavoro corrisponde alla *home*[12] dell'utente; essa può essere cambiata con il comando `cd` (da *Change Directory*) seguito dal pathname della directory in cui ci si vuole spostare, mentre la si può stampare a video con il comando `pwd` (da *Print Work Directory*. Si tenga presente poi che ogni directory contiene sempre almeno due voci: la directory `.` che fa riferimento a se stessa, e la directory `..`

[11] l'operazione di rendere visibili i file dentro un filesystem è chiamata così.

[12] ogni utente ha una sua directory personale nella quale può tenere i suoi file, che viene chiamata così.

che fa riferimento alla directory sovrastante, in questo modo anche con dei pathname relativi si possono fare riferimenti a directory poste in sezioni diverse dell'albero. Si noti come entrambe queste voci (dato che entrambe iniziano per `.`) siano *invisibili*.

La shell inoltre, quando deve passare dei pathname ai comandi che operano su file e directory (come `cd`, `cp`, ecc.) riconosce alcuni caratteri speciali, ad esempio il carattere `~` viene usato per indicare la home dell'utente corrente, mentre con `~username` si indica la home dell'utente `username`. Infine `cd` riconosce il carattere `-` che indica il ritorna alla precedente directory di lavoro, torneremo su questo con qualche dettaglio in più in sez. 2.1.3.

1.2.4 Il *Filesystem Hierarchy Standard*

Come per il processo `init`, che non è figlio di nessun altro processo e viene lanciato direttamente dal kernel, anche la directory radice non è contenuta in nessuna altra directory e, come accennato in sez. 1.1.1, 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.[13]

Un esempio di questa struttura ad albero, che al contempo ci mostra anche i contenuti delle directory principali, può essere ottenuto con il comando `tree`. Se chiamato senza parametri questo comando mostra l'albero completo a partire dalla directory corrente, scendendo in tutte le directory sottostanti; usando l'opzione `-L` si può specificare il numero massimo di livelli a cui scendere, per cui andando su `/` avremo qualcosa del tipo:

```
piccardi@oppish:~$ cd /
piccardi@oppish:/$ tree -L 2
.
|-- bin
|   |-- arch
...
|   |-- zmore
|   `-- znew
|-- boot
|   |-- System.map-2.4.20
...
|   |-- os2_d.b
|   `-- vmlinuz-2.4.20
|-- cdrom
|-- dev
|   |-- MAKEDEV -> /sbin/MAKEDEV
...
|   |-- xdb8
|   `-- zero
|-- etc
|   |-- GNUstep
...
|   |-- xpdf
|   `-- xpdfrc
|-- floppy
```

[13] in realtà essa può essere cambiata per i processi lanciati usando il comando `chroot`, in modo da restringere il loro accesso ai file contenuti in una sezione particolare dell'albero, il resto del sistema però continuerà a vedere la `/` originale.

```
|-- home
|   `-- piccardi
|-- initrd
|-- lib
|   |-- cpp -> /usr/bin/cpp-2.95
...
|   |-- modules
|   `-- security
|-- lost+found
|-- mnt
|   `-- usb
|-- opt
|-- proc
|   |-- 1
...
|   |-- uptime
|   `-- version
|-- root
|-- sbin
|   |-- MAKEDEV
...
|   |-- update-grub
|   `-- update-modules
|-- tmp
|   `-- ssh-XXBiWARl
|-- usr
|   |-- X11R6
|   |-- bin
|   |-- doc
|   |-- games
|   |-- include
|   |-- info
|   |-- lib
|   |-- local
|   |-- sbin
|   |-- share
|   `-- src
|-- var
|   |-- backups
|   |-- cache
|   |-- lib
|   |-- local
|   |-- lock
|   |-- log
|   |-- mail
|   |-- opt
|   |-- run
|   |-- spool
|   `-- tmp
`-- vmlinuz -> boot/vmlinuz-2.2.20-idepci
```