# Sistemi 13

Abbiamo visto che il file system è un dispositivo a blocchi. Può essere abbastanza comune voler partizionare un disco. Ci sono per fare ciò due standard:

* MBR (Master Boot Record), il più vecchio, che di fatto fa sì che il primo settore del disco contiene (oltre al boot del sistema) le informazioni sulla partizione del resto del disco, originariamente era pensato per permettere 4 partizioni primarie + altre estese.
* GPT (GUID Partition Table), permette di identificare le partizioni con un UUID. Questi UUID permettono di spostare come si vuole i dischi senza creare conflitti (?)

Nei sistemi Unix i dispositivi I/O sono speciali file considerati a caratteri o a blocchi (la maggior parte sono a blocchi). Questi file speciali sono identificati dal major number e dal minor number. Il major number corrisponde al tipo di dispositivo, mentre il minor number identifica le singole istanze.

Per esempio, se abbiamo dei dischi SATA, /dev/sda identificherà l’intero disco, mentre /dev/sda1, /dev/sda2 ecc. identificheranno le partizioni.

Questi dispositivi (ossia i file speciali) sono creati con mknod, da root (perché se un utente qualsiasi si potresse creare un file corrispondente al disco, potrebbe usarlo per invadere il disco). Leggere o scrivere su questi file vuol dire interagire direttamente col disco. In generale però non si vuole accedere direttamente al disco, per questo motivo vanno montati con mount. Se abbiamo un pendrive usb e vogliamo leggerne il contenuto su UNIX dobbiamo inserire il pendrive e poi montare il pendrive (agganciandolo nell’albero delle directory).

In Unix abbiamo un unico albero delle directory, quindi ogni disco esterno va montato sotto una cartella esistente, su Windows c’è una radice per ogni lettera (C: D: E: F: ecc.), quindi i dischi esterni vengono montati su una nuova lettera “dell’unità”. In Windows si inizia da C:, perché A: e B: sarebbero dedicati ai floppy disk.

Con l’aumentare del numero dei dispositivi supportati dal kernel e col fatto che ne posso agganciare nuovi a runtime, servirebbero avere sotto dev migliaia di file. Ma noi non abbiamo mai migliaia di dispositivi connessi contemporaneamente, per questo motivo il file speciale è creato da udev solo nel momento in cui il dispositivo viene collegato. Udev inoltre va a controllare e in caso carica i driver necessari per comunicare con la periferica.   
Per gestire in maniera non confusionaria le partizioni si usa l’UUID, specialmente perché tra un’accensione e l’altra potremmo modificare la posizione fisica dei nostri dischi, quindi se gli assegnassimo dei nomi e basta il kernel potrebbe non fare la stessa cosa ad ogni accensione.

Il file-system è una struttura dati che ci permette di utilizzare un dispositivo a blocchi in maniera più comoda (noi quando vogliamo creare un file lo facciamo e basta, senza posizionarlo sul disco). Sul disco quindi non ci sono solo i dati, ma anche i metadati (nome del file, ultima modifica, chi è il proprietario ecc.), che vanno memorizzati da qualche parte.   
Quando noi formattiamo un disco in realtà prepariamo le strutture dati in modo che corrispondano a un disco vuoto.

Noi vedremo una versione molto semplificata di un file system alla unix.

Quando utilizziamo i file a runtime a livello del kernel ci sono altre stutture dati, che stanno in memoria, che entrano in gioco. Un esempio è la tabella dei file descriptors.

Assumiamo di avere un disco , o partizione, di 64 blocchi da 4kb. (La granularità dei settori è di 512 byte, che però possono essere raggruppati in blocchi logici detti “clusters”, quindi avere blocchi da 4k ha senso). Le due ragioni per raggrupparli sono che si risparmia sul tempo di indirizzamento sui dischi veri non SSD e perché permette di indirizzare memorie più grandi con gli stessi indirizzi.

Useremo una grossa sezione del disco per i dati e un’altra porzione per i metadati.

In Unix, per ogni file c’è una struttura dati detta INode, presente nella tabella degli iNode. La Struttura di questa tabella ci dà un limite sul numero massimo di file sono memorizzabili sul disco (quando formattiamo il disco un parametro da mettere è quanti iNode poter memorizzare). Per ogni iNode dobbiamo sapere se è utilizzato o meno (in modo da poterli assegnare alla creazione di un nuovo file): per realizzare ciò si usa una bitMap (così facendo è più veloce trovare gli iNode liberi, perché ci evita di guardare in ogni iNode). Abbiamo anche una bitMap per i blocchi dei dati, che ci dice se sono occupati o liberi.  
Infine, viene messo un super blocco che serve ad identificare il tipo di file system e tutte le sue caratteristiche (numero di iNode, numero di blocchi dati ecc.): quando formattiamo il disco possiamo specificare il contenuto di questo super blocco, in modo da ottimizzare il contenuto del disco.

Che cosa c’è dentro l’iNode. Dentro ogni iNode ci sono tutti i metadati relativi a un file a parte il nome. C’è scritto qual è il tipo di file, che può essere: regular file, directory, symbolic link, FIFO, socket, character device block device (tutti i file classici, sequenze di byte di lunghezza arbitraria, sono regular files).

FIFO e socket sono sostanzialmente delle pipe con un nome e character device e block device sono le periferiche.

Oltre a questo, vengono specificati i UID e i GID di proprietario e gruppo, la dimensione in byte, la maschera dei bit relativa ai permessi, le date (creazione/modifica/ultimo accesso), il numero di hard link e il puntatore ai blocchi dati (questi puntatori sono i numeri di blocco che corrispondono a quelli che contengono i blocchi dati).

Tutti gli iNode sono lunghi uguali, il che sembra non avere senso se si pensa che ogni iNode ha un certo numero di puntatori ai blocchi: per risolvere ciò in ogni iNode c’è un numero di puntatori limitato (12) tali che puntano direttamente ai blocchi dati, dopodichè il (tredicesimo) puntatore punta a un blocco di puntatori che contiene 128 puntatori ai blocchi dati. Sebbene la doppia lettura sia costosa, il sistema di cache permette di ridurre il processo. Il 14simo puntatore punterà a un blocco di puntatori che puntanto a 128 blocchi di puntatori che puntano ai blocchi di dati. Il 15simo punterà a un blocco che punterà a 128 blocchi che punteranno a 128 blocchi ciascuno che punteranno ai blocchi di dati. Il procedimento è meno tremendo di quello che sembra per via della cache, che memorizza il puntatore diretto al blocco dati.  
Questo è anche il motivo per cui si possono avere file enormi che hanno grossi buchi (e che quindi occupano poco spazio) perché i puntatori ai blocchi dati del file sono quasi tutti a null.

Le directory sono file, esse contengono le associazioni tra nome e numero di iNode. Dentro ogni directory c’è un iNode . che contiene il nome di se stessa e un iNode .. che contiene il nome della directory parent.

Le associazioni nome-numero di iNode sono dette hard-link.

Tramite ln(1) possiamo aggiungere un nome a un file esistente, si ha un hard link (due nomi punteranno allo stesso numero di iNode, anche se si trovano in directory diverse, purchè sullo stesso filesystem).  
Invece, i link simbolici sono file che contengono un percorso e tale percorso non deve necessariamente esistere. Con rm si toglie un nome a un file, quindi se un file ha due nomi (per via di un hard link) serve fare rm di entrambi per far sì che il file/l’iNode sia considerato cancellato.

Anche se il numero di link va a 0, ma ci sono dei file aperti sull’iNode, l’iNode resta in giro finchè i file restano aperti; questo è un trucco che viene spesso usato per usare i file descriptor di file temporanei.

Se all’interno di una directory, si lancia il comando stat è possibile vedere le informazioni degli inode associati ai nomi della cartella.

Con ln -s si crea un link simbolico anziché un hard link.

I link simbolici permettono di fare collegamenti tra file system diversi (mentre con gli hard link no, perché questi danno nomi nuovi a un iNode che esiste già nel file system, il link simbolico invece può collegarsi anche a file system corrispondenti a dispositivi che vengono collegati e scollegati continuamente).

Nota che creare un hard link è ben diverso da creare una copia del file (abbiamo solo un altro nome che fa riferimento allo stesso file).

Per quanto riguarda le directory, i permessi rwx hanno senso solo se per x intendiamo l’accesso alla directory e se per r/w intendiamo la lettura e la modifica della directory (se non abbiamo l’accesso in scrittura di un file, ma lo abbiamo sulla directory possiamo fare rm del file dalla directory)

Path resolution: Nel PCB (process control block) ci sono iNode di root r e directory corrente c. Un percorso assoluto (che parte da r) inizia con “/”, un percorso relativo (che parte da c) inizia senza “/”.

Per ogni “componente” del percorso d (separata da / ) non finale, chiamiamola n:

si controlla se si hanno i permessi di accesso a d, se sì, si cerca n in d (sennò -> EACCESS). Se non si trova n -> EONET, altrimenti si recupera l’iNode corrispondente i. Se i è una directory si prosegue la ricerca da lì, se è un link simbolico, lo si risolve a partire da d finendo da qualche altra parte (se questa altra parte è una directory, si continua da lì, altrimenti -> ENOTDIR. Nota c’è un contatore per evitare loop infiniti ogni volta che si risolve un link simbolico). Se n non è una directory né un link simbolico -> ENOTDIR.

Per la componente finale non si pretende che sia una directory (ci sta bene che sia un file regolare).

. e .. hanno significato speciale: anche se il File system non li memorizzasse, e lo fa, l’algoritmo di traduzione li converte rispettivamente nella directory corrente e in quella parent. Attenzione che la root è considerata parent di se stessa, quindi se si chiama .. sulla root si resta sulla root.

Abbiamo già visto che una singola operazione logica va a modificare più strutture dati nel File System (se vogliamo creare una directory modifichiamo due bitMap, la tabella degli iNode e dovremmo allocare un blocco nello spazio dei blocchi dei dati). Cosa succede se durante queste operazioni manca la corrente? C’è un ordine delle operazioni che è migliore di altri per causare problematiche quando riparte il sistema? (per esempio se scriviamo prima sulle bitmap ma non sui blocchi iNode e dati quella parte di memoria ce la siamo giocata, perché risulta scritta ma non lo è, se invece scriviamo sui blocchi ma non sulle BitMap questi verranno presi da altri file successivamente). Per migliorare la situazione possiamo controllare l’integrità del file system con fsck.

Questo comando controlla che il super blocco sia “ragionevole” e che ci sia consistenza tra blocchi liberi e puntatori a file. Come risolvere se più inode puntano allo stesso blocco? Si può duplicare il blocco, dando a ognuno la sua copia. Se un blocco puntato risulta libero si imposta sulla bitmap che è occupa e se un blocco non puntato risulta usato si elimina il nome che lo punta.

Si potrebbe trovare un iNode con reference num. Maggiore di zero ma non trovare nessun nome, in tal caso gli si può dare un nome a babbo e lasciare che sia l’utente a dargliene uno serio.

Questo controllo prende parecchio tempo, quindi l’idea è di lanciare fsck solo quando il sistema è stato spento male (magari per calo di corrente). Per fare ciò si fa si che mentre si monta un file system si imposta un flag che lo identifica come “sporco” e lo si reimposta “pulito” quando lo si smonta (durante lo spegnimento). Se all’avvio si trova il flag a sporco si chiama l’fsck. In realtà non avviene neanche questo, al giorno d’oggi si usa il Journaling, un logging che riporta le modifiche che si volevano fare e si annota quando vengono completate, in questo modo quando qualcosa va male perché si spegne il dispositivo all’improvviso in realtà vengono ripetute solo le ultime operazioni che potrebbero non essere state completate (oppure viene riportato lo stato a Prima che fossero eseguite).

Nota: il fatto che il sistema si riporti automaticamente il File System a uno stato consistente non garantisce il salvataggio dei dati, anzi; è molto probabile che la maggior parte delle informazioni che volevamo salvare vadano perse comunque.