

08 - File System UNIX

UNIX file system: organizzazione logica

- Omogeneita': tutto e' file
- Tre categorie di file:
 - file **ordinari**
 - **direttori**
 - **dispositivi fisici**: file speciali (nel direttorio /dev)

Nome, i-number, i-node

Ad ogni file possono essere associati uno o piu' nomi simbolici, ma ad ogni file e' associato uno ed uno solo descrittore (i-node), univocamente identificato da un intero (i-number)

UNIX file system: organizzazione fisica

- Metodo di allocazione utilizzato in UNIX e' a **indice** (a piu' livelli di indirizzamento)
- Formattazione del disco in blocchi fisici (dimensione del blocco: 512B-5096B)
- Superficie del disco partizionata in 4 regioni:
 - boot block
 - super block
 - i-list
 - data blocks
- **Boot block** contiene le procedure di inizializzazione del sistema (da eseguire al bootstrap)
- **Super Block** fornisce
 - i limiti delle 4 regioni
 - il puntatore a una lista dei blocchi liberi
 - il puntatore a una lista degli i-node liberi
- **Data Blocks** - area del disco effettivamente disponibile per la memorizzazione dei file. Contiene
 - i blocchi allocati
 - i blocchi liberi (organizzati in una lista collegata)

i-List contiene una lista di tutti i descrittori (**i-node**) dei file normali, direttori e dispositivi presenti nel file system (accesso con l'indice **i-number**)

i-node

i-node e' il **descrittore** del file

Tra gli attributi contenenti nell'i-node vi sono:

- tipo di file
 - ordinario
 - direttorio
 - file speciale, per i dispositivi
- proprietario, gruppo (user-id, group-id)
- dimensione
- data
- 12 bit di protezione
- numero di link
- 13-15 indirizzi di blocchi (a seconda della realizzazione)

Indirizzamento

L'allocazione del file NON e' su blocchi fisicamente contigui. Nell'i-node sono contenuti **puntatori a blocchi** (ad esempio 13), del quali:

- i primi 10 indirizzi riferiscono blocchi di dati (indirizzamento diretto)
- 11esimo indirizzo: indirizzo di un blocco contenente a sua volta indirizzi di blocchi dati (primo livello di indirettezza)
- 12esimo indirizzo: secondo livello di indirettezza
- 13esimo indirizzo: terzo livello di indirettezza

Se: dimensione del blocco $512B = 0,5KB$

indirizzi di 32 bit (4 byte)

--> 1 blocco contiene 128 indirizzi

- 10 blocchi di dati sono accessibili direttamente
file di dimensioni minori di $10 \cdot 512B = 5KB$ sono accessibili direttamente
- 128 blocchi di dati sono accessibili con indirettezza singola (mediante il puntatore 11): $128 \cdot 512B = 64KB$
- $128 \cdot 128$ blocchi di dati sono accessibili con indirettezza doppia (mediante il puntatore 12):
 $128 \cdot 128 \cdot 512 = 8MB$
- $128 \cdot 128 \cdot 128$ blocchi di dati sono accessibili con indirettezza tripla (mediante il puntatore 13):
 $128 \cdot 128 \cdot 128 \cdot 512 = 1GB$

La dimensione massima del file realizzabile e' dell'ordine del GB

Dimensione massima = 1GB + 8MB + 64KB + 5KB

--> vantaggio aggiuntivo: l'accesso a file di piccole dimensioni e' piu' veloce rispetto al caso di file grandi

Breve parentesi su Linux file system

All'utente finale il file system Linux appare come una struttura ad albero gerarchico e obbedisce alla semantica UNIX. Internamente, il kernel di Linux e' capace di gestire differenti file system multipli fornendo un livello di astrazione uniforme: Virtual File System (VFS)

Linux VFS sfrutta:

- un insieme di descrittori che definiscono come un file debba apparire (inode object, file object, dile system object)
- un insieme di funzionalita' software per gestire questi oggetti

Linux Ext2fs

Ext2fs usa un meccanismo molto simile a quello standard UNIX per identificare i data block appartenenti a un file specifico

Le differenze principali riguardano le strategie di allocazione dei blocchi:

- Ext2fs usa default block size di 1KB, ma e' anche in grado di supportare blocchi da 2KB e 4Kb
- Ext2fs tenta di collocare i blocchi logicamente adiacenti di un file su blocchi fisicamente adiacenti su disco. In questo modo, e' teoricamente possibile realizzare una singola operazione di I/O che operi su diversi blocchi logicamente adiacenti

Linux: proc file system

- proc file system non serve a memorizzare dati, ma per funzioni tipicamente di monitoraggio
- E' uno pseudo-file system che fornisce accesso alla strutture dati di kernel mantenute da SO: i suoi contenuti sono tipicamente calcolati on demand
- proc realizza una struttura a directory e si occupa di mantenere i contenuti invocando le funzioni di monitoring correlate

Esempi: version, dma, stat/cpu, stat/swap, stat/process

Parentesi su MS file system NTFS

La struttura fondamentale per il file system NTFS (New Technology File System) di MS e' il volume

- basato su una partizione logica di disco
- puo' occupare una porzione di disco, un disco intero, o differenti porzioni su differenti dischi

Tutti i descrittori (metadati riguardanti il volume) sono memorizzati in un file normale

NTFS usa cluster come unita' di allocazione dello spazio su disco

- un cluster e' costituito da un numero di settori disco che e' potenza di 2
- frammentazione interna ridotta rispetto a 16-bit FAT perche' dimensione cluster minore
- Un file NTFS non e' una semplice sequenza di byte, come in MS-DOS o UNIX, ma un oggetto strutturato con attributi
- Ogni file NTFS e' descritto da una o piu' entry di un array memorizzato in un file speciale chiamato Master File Table (MFT)
- Ogni file NTFS su una unita' disco ha ID unico (64 bit)
- Lo spazio dei nomi di NTFS e' organizzato in una gerarchia di directory. Per motivi di efficienza, esiste un indice per ottimizzare l'accesso (B+ tree)

Directory

Anche le directory sono rappresentate nel file system da file

- Ogni file-directory contiene un insieme di record logici con struttura
- Ogni record rappresenta un file appartenente alla directory
 - per ogni file (o directory) appartenenza alla directory considerata, viene memorizzato il suo nome relativo, a cui viene associato il relativo **i-number** (che lo identifica univocamente)

Gestione file system in UNIX

Quali sono i meccanismi di accesso al file system?

Come vengono realizzati?

Vanno considerati

- strutture dati di sistema per il supporto all'accesso e alla gestione di file
- principali system call per l'accesso e la gestione di file

Gestione file system: concetti generali

- Accesso sequenziale
- Assenza di strutturazione: **file = sequenza di byte (stream)**
- Posizione corrente: **I/O Pointer**
- Varie **modalita'** di accesso (lettura, scrittura, lettura/scrittura, ...)
- Accesso subordinato all'operazione di **apertura**

File descriptor

- A ogni processo e' associata una **tabella dei file aperti** di dimensione limitata (attorno al centinaio di elementi)
- Ogni elemento della tabella rappresenta un file aperto dal processo ed e' individuato da un indice intero: **file descriptor**
- I file descriptor **0, 1, 2** individuano rispettivamente standard input, output, error (aperti automaticamente)
- La tabella dei file aperti del processo e' allocata nella sua **user structure**

Struttura dati kernel

Per realizzare l'accesso ai file, SO utilizza **due strutture dati** globali, allocate nell'area dati del kernel

- la **tabella dei file attivi**: per ogni aperto, contiene una copia del suo i-node --> piu' efficienti le operazioni su file evitando accessi al disco per ottenere attributi dei file acceduti
- la **tabella dei file aperti di sistema**: ha un elemento per ogni operazione di apertura relativa a file aperti (e non ancora chiusi). Ogni elemento contiene:
 - I/O pointer, posizione corrente all'interno del file
 - un puntatore all'i-node del file nella tabella dei file attivi

--> se due processo aprono separatamente lo stesso file F, la tabella conterra' due elementi distinti associati a F

Riassumendo:

- **tabella dei file aperti di processo**: nella user area del processo, contiene un elemento per ogni file aperto dal processo
- **tabella dei file aperti di sistema**: in area di SO, contiene un elemento per ogni sessione di accesso a file nel sistema
- **tabella dei file attivi**: in area di SO, contiene un elemento per ogni file aperto nel sistema

Tabella dei file aperti di sistema

Un elemento per ogni "apertura" di file: a processi che accedono allo stesso file corrispondono entry distinte.

Ogni elemento contiene il puntatore alla posizione corrente (I/O pointer)

--> piu' processi possono accedere contemporaneamente allo stesso file, ma hanno **I/O pointer distinti**

Tabella dei file attivi

L'operazione di **apertura** provoca la copia dell'i-node in memoria centrale (se il file non e' gia' in uso).

La tabella dei file attivi contiene gli i-node di tutti i file aperti

Il numero degli elementi e' pari al numero dei file aperti (anche da piu' di un processo)

Gestione file system UNIX: system call

UNIX permette ai processi di accedere a file, mediante un insieme di system call, tra le quali:

- apertura/creazione: `open()`, `creat()`
- chiusura: `close()`
- lettura: `read()`
- scrittura: `write()`
- cancellazione: `unlink()`
- linking: `link()`
- accesso diretto: `lseek()`

System Call: apertura di file

L'apertura di un file provoca:

- **inserimento di un elemento** (individuato da un file descriptor) nella prima posizione libera della tabella dei file aperti del processo
- **inserimento di un nuovo record nella tabella dei file aperti di sistema**
- la **copia dell'i-node nella tabella dei file attivi** (solo se il file non e' gia' in uso)

Apertura di un file: `open()`

Per aprire un file:

```
int open(char nomefile[], int flag, [int mode]);
```

- - `nomefile` e' il nome del file (relativo o assoluto)
 - `flag` esprime il modo di accesso; ad esempio `O_RDONLY`, per accesso in lettura, `O_WRONLY`, per accesso in scrittura
 - `mode` e' un parametro richiesto soltanto se l'apertura determina la creazione del file (flag `O_CREAT`): in tal caso, `mode` specifica i bit di protezione
- Valore restituito dalla `open()` del file descriptor associato al file, -1 in caso di errore
 - se `open()` ha successo, il file viene aperto nel modo richiesto e I/O pointer posizionato sul primo elemento (tranne nel caso di `O_APPEND`)

Modi di apertura (definiti in `<fcntl.h>`)

- `O_RDONLY` (=0), accesso in lettura
- `O_WRONLY` (=1), accesso in scrittura

- O_APPEND (=2), accesso in scrittura in modalita' append (in fondo al file), sempre da associare a O_WRONLY

Inoltre e' possibile abbinare ai tre metodi precedenti, altri modi (mediante il connettore |):

- O_CREAT, per accesso in scrittura. Se il file non esiste, viene creato, e' necessario fornire il parametro mode, per esprimere i bit di protezione
- O_TRUNC, per accesso in scrittura: la lunghezza del file viene troncata a 0

Apertura del file: creat()

Per creare un file:

```
int creat(char nomefile[], int mode);
```

- - nomefile e' il nome del file (relativo o assoluto) da creare
 - mode e' necessario e specifica i 12 bit di protezione per il nuovo file
- Valore restituito da creat() e' il file descriptor associato al file, -1 in caso di errore
- se creat() ha successo, il file viene aperto in scrittura e I/O pointer posizionato sul primo elemento (se file esiste, viene cancellato e riscritto da capo)

Chiusura di file: close()

Per chiudere un file aperto:

```
int close(int fd);
```

fd e' il file descriptor del file da chiudere

Restituisce l'esito della operazione (0 in caso di successo, <0 in caso di insuccesso)

Se close() ha successo:

- file memorizzato sul disco
- eliminato l'elemento di indice fd dalla tabella dei file aperti di processo
- eventualmente eliminati gli elementi dalla tabella dei file di sistema e dei file attivi

System call: lettura e scrittura

Caratteristiche:

- accesso mediante file descriptor
- ogni operazione di lettura (o scrittura) agisce sequenzialmente sul file, a partire dalla posizione corrente di I/O pointer

- possibilita' di alternare operazioni di lettura e scrittura
- atomicita' della singola operazione
- operazioni sincrone, cioe' con attesa del completamento logico dell'operazione

Lettura di file: read()

```
int read(int fd, char *buf, int n);
```

- - fd - file descriptor del file
 - buf - area in cui trasferire i byte letti
 - n - numero di caratteri da leggere
- In caso di successo, restituisce un intero ($\leq n$) che rappresenta il numero di caratteri effettivamente letti

Per indicare da tastiera (in fase di input) la volonta' di terminare il file, <CTRL-D>

Se read() ha successo:

- a partire dal valore corrente di I/O pointer, vengono letti al piu' n bytes dal file fd e memorizzati all'indirizzo buff
- I/O pointer viene spostato avanti di n bytes

Scrittura file: write()

```
int write(int fd, char *buf, int n);
```

- - fd - file descriptor del file
 - buf - area da cui trasferire i byte scritti
 - n - numero di caratteri da scrivere

In caso di successo, restituisce un intero positivo ($= n$) che rappresenta il numero di caratteri effettivamente scritti

Accesso diretto: lseek()

Per spostare I/O pointer:

```
lseek(int fd, int offset, int origine);
```

- - fd - file descriptor del file
 - offset - spostamento (in byte) rispetto all'origine

- origine - puo' valere
 - 0: inizio file (SEEK_SET)
 - 1: posizione corrente (SEEK_CUR)
 - 2: fine file (SEEK_END)

In caso di successo, restituisce un intero positivo che rappresenta la nuova posizione

Cancellazione di file: unlink()

Per cancellare un file, o decrementare il numero dei suoi link

```
int unlink(char *name);
```

- - name - nome del file
 - ritorna 0 se OK, altrimenti -1

In generale, l'effetto della system call unlink() e' decrementare di 1 il numero di link del file dato (nell'i-node); solo nel caso in cui il numero dei link risulti 0, allora il file viene effettivamente cancellato

Aggiungere nomi a file esistenti: link()

Per aggiungere un link a un file esistente:

```
int link(char *oldname, char *newname);
```

- - oldname - nome del file esistente
 - newname - nome associato al nuovo link()

link()

- incrementa il numero dei link associato al file nell'i-node
- aggiorna il direttorio (aggiunta di un nuovo elemento)
- ritorna 0 in caso di successo, -1 se fallisce

ad esempio, fallisce se:

- oldname non esiste
- newname esiste gia' (non viene sovrascritto)
- oldname e newname appartengono a unita' disco diverse (in questo caso, usare link simbolici mediante symlink)

Diritti di accesso a file: chmod()

Per modificare i bit di protezione di un file:

```
int chmod(char *pathname, char *newmode);
```

- pathname - nome del file
- newmode - contiene i nuovi diritti

Ad esempio: `chmod("file.txt", "0777")`

```
chmod("file.txt", "o-w")
```

```
chmod("file.txt", "g+x")
```

Diritti di accesso a file: chown()

Per cambiare il proprietario e il gruppo di un file:

```
int chown(char *pathname, int owner, int group);
```

- pathname - nome del file
- owner - uid del nuovo proprietario
- group - gid del gruppo

Cambia proprietario/gruppo del file

Gestione directory

Alcune system call a disposizione in C/UNIX per la gestione delle directory

- chdir(): per cambiare directory (come comando shell cd)
- opendir(), closedir(): apertura e chiusura di directory
- readdir(): lettura di directory

La system call sopra fanno uso di tipi astratti per la descrizione della struttura dati directory e sono indipendenti da come il direttorio viene realizzato (BSD, System V, Linux)

Per effettuare un cambio di directory

```
int chdir(char *nomedir);
```

- - nomedir - nome della directory in cui entrare
 - restituisce 0 in caso di successo (cioè cambio di directory avvenuto), -1 in caso di fallimento
- Analogamente ai file normali, lettura/scrittura di una directory possono avvenire solo dopo l'operazione di apertura opendir()
- Apertura restituisce un puntatore a DIR:
 - DIR è un tipo di dato astratto predefinito (<dirent.h>) che consente di riferire (mediante puntatore) una directory aperta

Apertura e chiusura di directory: opendir(), closedir()

Per aprire un direttorio

```
#include <dirent.h>
DIR *opendir (char *nomedir);
```

nomedir - nome del direttorio da aprire

restituisce un valore di tipo puntatore a DIR:

- diverso da NULL se l'apertura ha successo: per gli accessi successivi, si impieghera' questo valore per riferire il direttorio
- altrimenti restituisce NULL

Chiusura del direttorio riferito al puntatore dir:

```
#include <dirent.h>
int closedir (DIR *dir);
```

Restituisce 0 in caso di successo, -1 altrimenti

Lettura di una directory

Una directory aperta puo' essere letta con readdir():

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *descr;
descr = readdir(DIR *dir);
```

dir - puntatore alla directory da leggere (valore restituito da opendir)

Restituisce:

- - un puntatore diverso da NULL se la lettura ha avuto successo
 - altrimenti restituisce NULL (in caso di insuccesso)
- In caso di successo, le readdir() legge un elemento dalla directory data e lo memorizza all'indirizzo puntato da descr
- descr punta ad una struttura di tipo dirent (dichiarata in dirent.h)

L'elemento nella directory: dirent

Il generico elemento (file o directory) in una directory e' rappresentato da un record di tipo dirent:

```
struct dirent {
    long d_ino; /* i-number */
    off_t d_off; /* offset del prossimo */
    unsigned short d_reclen; /* lunghezza del record */
    unsigned short d_namelen; /* lunghezza del nome */
    char d_name[1]; /* nome del file */
}
```

La stringa che parte da d_name rappresenta il nome del file (o directory) nella directory aperta; d_namelen la lunghezza del nome.

--> possibilita' di nomi con lunghezza variabile

Gestione di directory: creazione

Creazione di una directory

```
int mkdir (char *pathname, int mode);
```

- pathname - nome del direttorio da creare
- mode - esprime i bit di protezione

Restituisce il valore 0 in caso di successo, altrimenti -1.

In caso di successo, crea e inizializza una directory con il nome e i diritti specificati. Vengono sempre creati i file:

- . (link alla directory corrente)
- .. (link alla directory padre)