# Il File System

Ivan Scagnetto

Università di Udine — Facoltà di Scienze MM.FF.NN.

A.A. 2011-2012

Copyright ©2000–04 Marino Miculan (miculan@dimi.uniud.it)

La copia letterale e la distribuzione di questa presentazione nella sua integrità sono permesse con qualsiasi mezzo, a condizione che questa nota sia riprodotta.

Alcune necessità dei processi:

- memorizzare e trattare grandi quantità di informazioni (maggiori della quantità di memoria principale),
- più processi devono avere la possibilità di accedere alle informazioni in modo concorrente e coerente, nello spazio e nel tempo,
- si deve garantire integrità, indipendenza, persistenza e protezione dei dati.

L'accesso diretto ai dispositivi di memorizzazione di massa (come visto nella gestione dell'I/O) non è sufficiente.

# I File

La soluzione sono i file (archivi):

- File = insieme di informazioni correlate a cui è stato assegnato un nome.
- Un file è la più piccola porzione unitaria di memoria logica secondaria allocabile dall'utente o dai processi di sistema.
- La parte del S.O. che realizza questa astrazione, nascondendo i dettagli implementativi legati ai dispositivi sottostanti, è il file system.
- Esternamente, il file system è spesso l'aspetto più visibile di un S.O. (S.O. documentocentrici): come si denominano, manipolano, accedono, quali sono le loro strutture, i loro attributi, ecc.
- Internamente, il file system si appoggia alla gestione dell'I/O per implementare ulteriori funzionalità.

# Attributi dei file (metadata)

**Nome** identificatore del file. L'unica informazione umanamente leggibile.

**Tipo** nei sistemi che supportano più tipi di file. Può far parte del nome.

**Locazione** puntatore alla posizione del file sui dispositivi di memorizzazione.

**Dimensioni** attuale, ed eventualmente massima consentita.

**Protezioni** controllano chi può leggere, modificare, creare, eseguire il file.

**Identificatori dell'utente** che ha creato/possiede il file.

**Varie date e timestamp** di creazione, modifica, aggiornamento info...

Queste informazioni (metadati: dati sui dati) sono solitamente mantenute in apposite strutture (directory) residenti in memoria secondaria.

# Attributi dei file (metadata)

| Attribute | Meaning |
| --- | --- |
| Protection | Who can access the file and in what way |
| Password | Password needed to access the file |
| Creator | ID of the person who created the file |
| Owner | Current owner |
| Read-only flag | 0 for read/write; 1 for read only |
| Hidden flag | 0 for normal; 1 for do not display in listings |
| System flag | 0 for normal files; 1 for system file |
| Archive flag | 0 for has been backed up; 1 for needs to be backed up |
| ASCII/binary flag | 0 for ASCII file; 1 for binary file |
| Random access flag | 0 for sequential access only; 1 for random access |
| Temporary flag | 0 for normal; 1 for delete file on process exit |
| Lock flags | 0 for unlocked; nonzero for locked |
| Record length | Number of bytes in a record |
| Key position | Offset of the key within each record |
| Key length | Number of bytes in the key field |
| Creation time | Date and time the file was created |
| Time of last access | Date and time the file was last accessed |
| Time of last change | Date and time the file has last changed |
| Current size | Number of bytes in the file |
| Maximum size | Number of bytes the file may grow to |

# Denominazione dei file

- I file sono un meccanismo di astrazione, quindi ogni oggetto deve essere denominato.
- Il nome viene associato al file dall'utente, ed è solitamente necessario (ma non sufficiente) per accedere ai dati del file.
- Le regole per denominare i file sono fissate dal file system, e sono molto variabili:
  - lunghezza: fino a 8, a 32, a 255 caratteri
  - tipo di caratteri: solo alfanumerici o anche speciali; e da quale set? ASCII, ISO-qualcosa, Unicode?
  - case sensitive, insensitive, preserving
  - contengono altri metadati? ad esempio, il tipo?

# Tipi dei file — FAT: name.extension

| Tipo | Estensione | Funzione |
|---|---|---|
| Eseguibile | exe, com, bin o nessuno | programma pronto da eseguire, in liguaggio macchina |
| Oggetto | obj, o | compilato, in linguaggio macchina, non linkato |
| Codice sorgente | c, p, pas, f77, asm, java | codice sorgente in diversi linguaggi |
| Batch | bat, sh | script per l'interprete comandi |
| Testo | txt, doc | documenti, testo |
| Word processor | wp, tex, doc | svariati formati |
| Librerie | lib, a, so, dll | librerie di routine |
| Grafica | ps, dvi, gif | FILE ASCII o binari |
| Archivi | arc, zip, tar | file correlati, raggruppati in un file, a volte compressi |

# Tipi dei file — Unix: nessuna assunzione

Unix non forza nessun tipo di file a livello di sistema operativo: non ci sono metadati che mantengono questa informazione.
Tipo e contenuto di un file slegati dal nome o dai permessi.
Sono le applicazioni a sapere di cosa fare per ogni file (ad esempio, i client di posta usano i MIME-TYPES).
È possibile spesso "indovinare" il tipo ispezionando il contenuto alla ricerca dei magic numbers: utility `file`

```
$ file iptables.sh risultati Lucidi
iptables.sh: Bourne shell script text executable
risultati: ASCII text
Lucidi: PDF document, version 1.2
```

# Tipi dei file — MacOS classico: molti metadati

Nel MacOS Classic ogni file è composto da 3 componenti:

- data fork: sequenza non strutturata, simile a quelli Unix o DOS;
- resource fork: strutturato, contiene codice, icone, immagini, etichette, dialoghi,. . .
- info: metadati sul file stesso, tra cui applicativo creatore e tipo.

L'operazione di apertura (doppio click) esegue l'applicativo indicato nelle info.

# Struttura dei file

- In genere, un file è una sequenza di bit, byte, linee o record il cui significato è assegnato dal creatore.
- A seconda del tipo, i file possono avere struttura
  - nessuna: sequenza di parole, byte,
  - sequenza di record: linee, blocchi di lunghezza fissa/variabile,
  - strutture più complesse: documenti formattati, archivi (ad albero, con chiavi, ...), eseguibili rilocabili (ELF, COFF),
  - i file strutturati possono essere implementati con quelli non strutturati, inserendo appropriati caratteri di controllo.
- Chi impone la struttura: due possibilità
  - Il sistema operativo: specificato il tipo, viene imposta la struttura e modalità di accesso. Più astratto.
  - L'utente: tipo e struttura sono delegati al programma, il sistema operativo implementa solo file non strutturati. Più flessibile.

# Operazioni sui file

**Creazione:** due passaggi: allocazione dello spazio sul dispositivo, e collegamento di tale spazio al file system.

**Cancellazione:** staccare il file dal file system e deallocare lo spazio assegnato al file.

**Apertura:** caricare alcuni metadati dal disco nella memoria pricipale, per velocizzare le chiamate seguenti.

**Chiusura:** deallocare le strutture allocate nell'apertura.

**Lettura:** dato un file e un puntatore di posizione, i dati da leggere vengono trasferiti dal media in un buffer in memoria.

**Scrittura:** dato un file e un puntatore di posizione, i dati da scrivere vengono trasferiti sul media.

**Append:** versione particolare di scrittura.

**Riposizionamento (seek):** non comporta operazioni di I/O.

**Troncamento:** azzerare la lunghezza di un file, mantenendo tutti gli altri attributi.

**Lettura dei metadati:** leggere le informazioni come nome, timestamp, ecc.

**Scrittura dei metadati:** modificare informazioni come nome, timestamp, protezione, ecc.

# Tabella dei file aperti

Queste operazioni richiedono la conoscenza delle informazioni contenute nelle directory. Per evitare di accedere continuamente alle dir, si mantiene in memoria una tabella dei file aperti. Due nuove operazioni sui file:

- Apertura: allocazione di una struttura in memoria (file descriptor o file control block) contenente le informazioni riguardanti un file.
- Chiusura: trasferimento di ogni dato in memoria al dispositivo, e deallocazione del file descriptor.

A ciascun file aperto si associa

- Puntatore al file: posizione raggiunta durante la lettura/scrittura.
- Contatore dei file aperti: quanti processi stanno utilizzando il file.
- Posizione sul disco.

# Metodi di accesso: accesso sequenziale

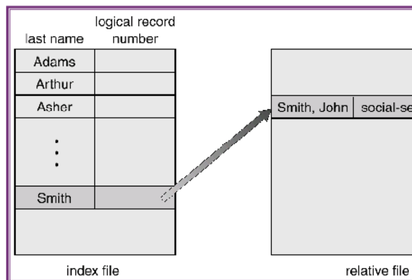
- Un puntatore mantiene la posizione corrente di lettura/scrittura
- Si può accedere solo progressivamente, o riportare il puntatore all'inizio del file.
- Operazioni:
  *read next*
  *write next*
  *reset*
  no *read* dopo l'ultimo *write*
  (*rewrite*)
- Adatto a dispositivi intrinsecamente sequenziali (p.e., nastri)

# Metodi di accesso: accesso diretto

- Il puntatore può essere spostato in qualunque punto del file
- Operazioni:
  *read n*
  *write n*
  *seek n*
  *read next*
  *write next*
  *rewrite n*
  $n$ = posizione relativa a quella attuale
- L'accesso sequenziale viene simulato con l'accesso diretto.
- Usuale per i file residenti su device a blocchi (p.e., dischi).

# Metodi di accesso: accesso indicizzato

- Un secondo file contiene solo parte dei dati, e puntatori ai blocchi (record) del vero file
- La ricerca avviene prima sull'indice (corto), e da qui si risale al blocco

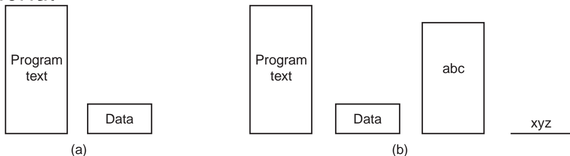



- Implementabile a livello applicazione in termini di file ad accesso diretto
- Usuale su mainframe (IBM, VMS), database. . .

```
/* File copy program. Error checking and reporting is minimal. */

#include <sys/types.h>                  /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);       /* ANSI prototype */

#define BUF_SIZE 4096                   /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700                /* protection bits for output file */

int main(int argc, char *argv[])
{
     int in_fd, out_fd, rd_count, wt_count;
     char buffer[BUF_SIZE];

     if (argc != 3) exit(1);            /* syntax error if argc is not 3 */

     /* Open the input file and create the output file */
     in_fd = open(argv[1], O_RDONLY);   /* open the source file */
     if (in_fd < 0) exit(2);            /* if it cannot be opened, exit */
     out_fd = creat(argv[2], OUTPUT_MODE);  /* create the destination file */
     if (out_fd < 0) exit(3);           /* if it cannot be created, exit */

     /* Copy loop */
     while (TRUE) {
          rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
     if (rd_count <= 0) break;          /* if end of file or error, exit loop */
          wt_count = write(out_fd, buffer, rd_count); /* write data */
          if (wt_count <= 0) exit(4);   /* wt_count <= 0 is an error */
     }

     /* Close the files */
     close(in_fd);
     close(out_fd);
     if (rd_count == 0)                 /* no error on last read */
          exit(0);
     else
          exit(5);                      /* error on last read */
}
```

# File mappati in memoria

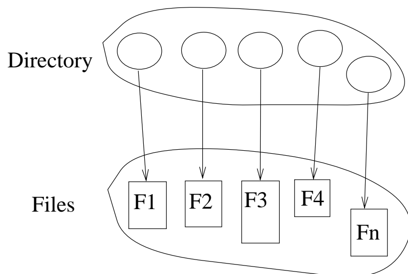
- Semplificano l'accesso ai file, rendendoli simili alla gestione della memoria.




- Relativamente semplice da implementare in sistemi segmentati (con o senza paginazione): il file viene visto come area di swap per il segmento mappato.
- Non servono chiamate di sistema `read` e `write`, solo una `mmap`.
- Problemi:
  - lunghezza del file non nota al sistema operativo,
  - accesso condiviso con modalità diverse,
  - lunghezza del file maggiore rispetto alla dimensione massima dei segmenti.

# Directory

- Una directory è una collezione di nodi contenente informazioni sui file (metadati).
- Sia la directory che i file risiedono su disco.
- Operazioni su una directory:
  - ricerca di un file,
  - creazione di un file,
  - cancellazione di un file,
  - listing,
  - rinomina di un file,
  - navigazione del file system.



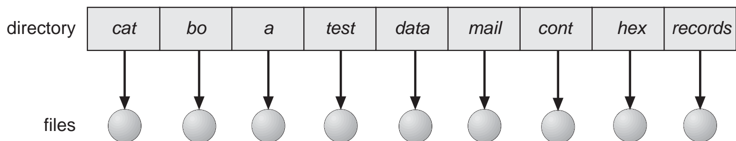
# Organizzazione logica delle directory

Le directory devono essere organizzate per ottenere

- efficienza: localizzare rapidamente i file
- nomi mnemonici: comodi per l'utente
  - file differenti possono avere lo stesso nome
  - più nomi possono essere dati allo stesso file
- Raggruppamento: file logicamente collegati devono essere raccolti assieme (e.g., i programmi in C, i giochi, i file di un database, . . . )

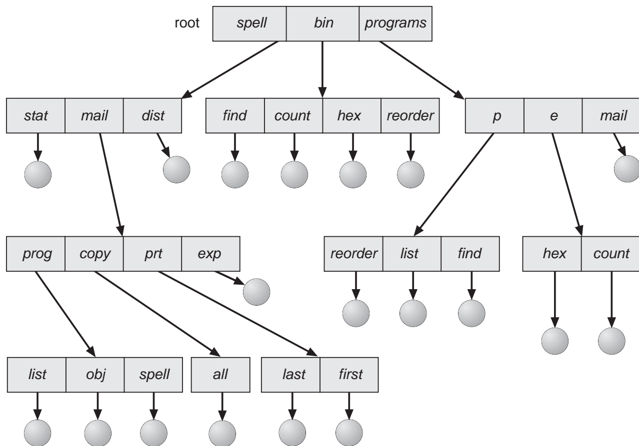
# Tipi di directory: unica (“flat”)

- Una sola directory per tutti gli utenti



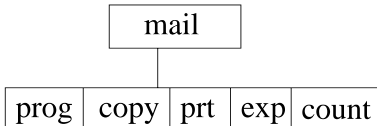

- Problema di raggruppamento e denominazione
- Obsoleta
- Variante: a due livelli (una directory per ogni utente)

# Tipi di directory: ad albero



# Directory ad albero (cont.)

- Ricerca efficiente
- Raggruppamento
- Directory corrente (working directory): proprietà del processo
  - **cd** /home/miculan/src/C
  - **cat** hw.c
- Nomi assoluti o relativi
- Le operazioni su file e directory (lettura, creazione, cancellazione, ...) sono relative alla directory corrente. Esempio: se la dir corrente è `/spell/mail`,
  **mkdir** `count`
  crea la situazione corrente:




- Cancellando `mail` si cancella l'intero sottoalbero.

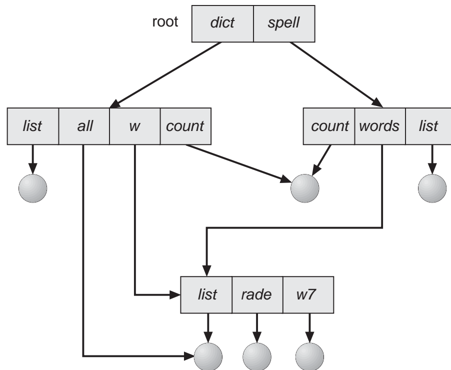
# Directory a grafo aciclico (DAG)

File e sottodirectory possono essere condivise da più directory

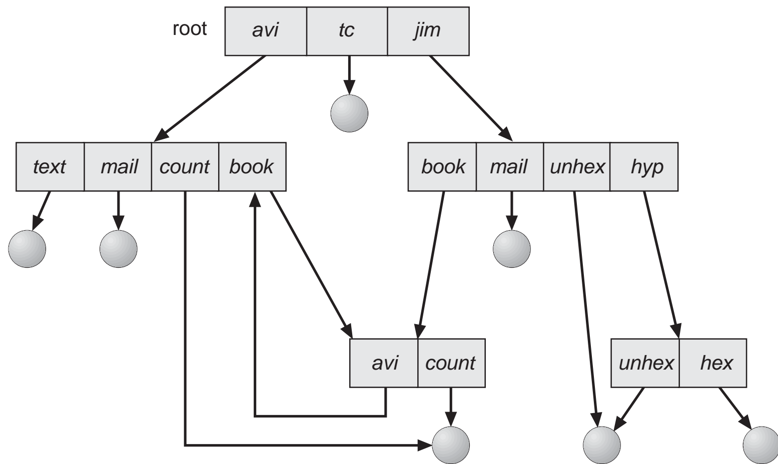
Due nomi differenti per lo stesso file (aliasing)

Possibilità di puntatori "dangling". Soluzioni:

- Puntatori all'indietro, per cancellare tutti i puntatori. Problematici perché la dimensione dei record nelle directory è variabile.
- Puntatori a daisy chain
- Contatori di puntatori per ogni file (UNIX)



# Directory a grafo



# Directory a grafo (cont.)

I cicli sono problematici per la

- Visita: algoritmi costosi per evitare loop infiniti
- Cancellazione: creazione di garbage

Soluzioni:

- Permettere solo link a file (UNIX per i link hard)
- Durante la navigazione, limitare il numero di link attraversabili (UNIX per i simbolici)
- Garbage collection (costosa!)
- Ogni volta che un link viene aggiunto, si verifica l'assenza di cicli. Algoritmi costosi.

# Protezione

- Importante in ambienti multiuser dove si vuole condividere file
- Il creatore/possessore (non sempre coincidono) deve essere in grado di controllare
  - cosa può essere fatto
  - e da chi (in un sistema multiutente)
- Tipi di accesso soggetti a controllo (non sempre tutti supportati):
  - Read
  - Write
  - Execute
  - Append
  - Delete
  - List

Sono il metodo di protezione più generale

| domain \ object | $F_1$ | $F_2$ | $F_3$ | printer |
|---|---|---|---|---|
| $D_1$ | read | | read | |
| $D_2$ | | | | print |
| $D_3$ | | read | execute | |
| $D_4$ | read<br>write | | read<br>write | |

- per ogni coppia (processo,oggetto), associa le operazioni permesse
- matrice molto sparsa: si implementa come
  - access control list: ad ogni oggetto, si associa chi può fare cosa.
    Sono implementate da alcuni UNIX (e.g., getfacl(1) e setfacl(1) su Solaris)
  - capability tickets: ad ogni processo, si associa un insieme di tokens che indicano cosa può fare

# Modi di accesso e gruppi in UNIX

Versione semplificata di ACL.

- Tre modi di accesso: **r**ead, **w**rite, **e**xecute
- Tre classi di utenti, per ogni file

| | | | | RWX |
|---|---|---|---|---|
| a) | owner access | 7 | ⇒ | 1 1 1 |
| b) | groups access | 6 | ⇒ | 1 1 0 |
| c) | public access | 1 | ⇒ | 0 0 1 |

- Ogni processo possiede UID e GID, con i quali si verifica l'accesso

# Modi di accesso e gruppi in UNIX

- Per limitare l'accesso ad un gruppo di utenti, si chiede al sistemista di creare un gruppo apposito, sia $G$, e di aggiungervi gli utenti.
- Si definisce il modo di accesso al file o directory

owner group public

**chmod** 761 *game*

- Si assegna il gruppo al file:

**chgrp** $G$ *game*

# Effective User e Group ID

- In UNIX, il dominio di protezione di un processo viene ereditato dai suoi figli, e viene impostato al login
- In questo modo, tutti i processi di un utente girano con il suo UID e GID.
- Può essere necessario, a volte, concedere temporaneamente privilegi speciali ad un utente (es: ps, lpr, . . . )
  - Effective UID e GID (EUID, EGID): due proprietà extra di tutti i processi (stanno nella U-structure).
  - Tutti i controlli vengono fatti rispetto a EUID e EGID
  - Normalmente, EUID=UID e EGID=GID
  - L'utente `root` può cambiare questi parametri con le system call setuid(2), setgid(2), seteuid(2), setegid(2)

# Setuid/setgid bit

- l'Effective UID e GID di un processo possono essere cambiati per la durata della sua esecuzione attraverso i bit **setuid** e **setgid**
- Sono dei bit supplementari dei file eseguibili di UNIX

```
miculan@coltrane:Lucidi$ ls -l
/usr/bin/lpr -r-sr-sr-x   1 root     lp          15608 Oct 23
07:51 /usr/bin/lpr* miculan@coltrane:Lucidi$
```

- Se **setuid** bit è attivo, l'EUID di un processo che esegue tale programma diventa lo stesso del possessore del file
- Se **setgid** bit è attivo, l'EGID di un processo che esegue tale programma diventa lo stesso del possessore del file
- I real UID e GID rimangono inalterati

# Setuid/setgid bit (cont.)

- Si impostano con il chmod

```
miculan@coltrane:C$ ls -l a.out
-rwxr-xr-x 1 miculan  ricerca 12045 Feb 28 12:11 a.out*
miculan@coltrane:C$ chmod 2755 a.out
miculan@coltrane:C$ ls -l a.out
-rwxr-sr-x   1 miculan  ricerca 12045 Feb 28 12:11 a.out*
miculan@coltrane:C$ chmod 4755 a.out
miculan@coltrane:C$ ls -l a.out
-rwsr-xr-x   1 miculan  ricerca 12045 Feb 28 12:11 a.out*
miculan@coltrane:C$
```