La pipe ("tubo" in inglese)

Meccanismo di comunicazione fra processi

Il più vecchio, condiviso da tutti i sistemi Unix

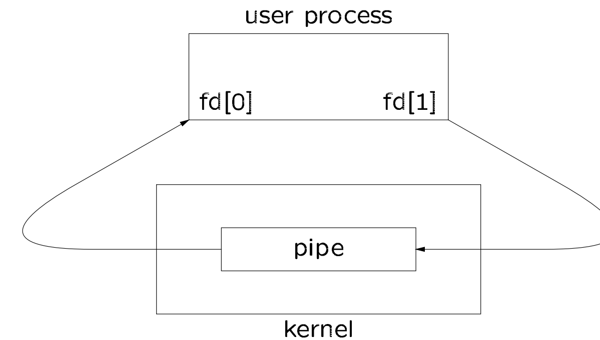
Ha due limitazioni:
- half-duplex: i dati scorrono in una sola direzione
- può essere usato solo fra processi che condividono un antenato

int pipe(int fd[2]);

Restituisce 0 se ha successo, -1 per errore

Restituisce due file descriptor connessi alla pipe nell'array *fd*
- fd[0] legge dalla pipe
- fd[1] scrive nella pipe

Visualizzare la pipe



Una pipe ha senso solo se dopo faccio una fork

Dopo la fork, normalmente uno dei processi chiude fd[0] e l'altro chiude fd[1]

Esempio: trasmissione dati dal figlio al genitore

```
int main() {
  int n, fd[2];
  pid_t pid;
  char buf[MAXLINE];

  if (pipe(fd) < 0) exit(EXIT_FAILURE);
  if ((pid = fork()) < 0) exit(EXIT_FAILURE);
  if (pid > 0) {
    close(fd[0]);
    write(fd[1], "Hello world\n", 12);
  } else {
    close(fd[1]);
    n = read(fd[0], buf, MAXLINE);
    write(1, buf, n);
  }
}
```


Esempio: la redirezione della shell (iii)

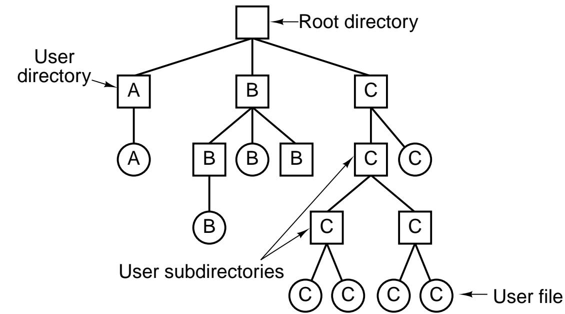
```
$ ls | less
```

La shell la implementa con

```
if (fork() > 0) {
  // siamo nel processo che sarà ‘‘ls’’
  pipe(fd);
  if (fork() > 0) {
    // siamo nel processo che sarà ‘‘less’’
    close(0);
    close(fd[1]);
    dup(fd[0]);
    close(fd[0]);
    execve("less", ...);
  } else {
    close(1);
    close(fd[0]);
    dup(fd[1]);
    close(fd[1]);
    execve("ls", ...);
  }
} else {
  // siamo nel processo shell
  waitpid(...);
}
```

```
shell interattiva
 |
fork-----------------+
 |                   |
wait                pipe
 .                   |
 .                  fork----------------+
 .                   |                  |
 .                  close(fd[0])       close(fd[1])
 .                  close(1)           close(0)
 .                  dup(fd[1]);        dup(fd[0])
 .                  close(fd[1]);      close(fd[0]);
 .                   |                  |
 .                  exec("ls",...);    exec("less", ...);
 .                   |                  |
 |<-------------------<----------------+
 |
```


Directories



Tutti i S.O. moderni usano directory gerarchiche

Path Names

Un *path name* specifica un file

Assoluto:

- specifica tutte le directory dalla radice fino al file
- es. Unix: /home/matteo/pippo.txt
- es. Windows: C:\Documents\Musica\zot.mp3
  (in Windows ci sono più radici; in Unix una sola)

Relativo:

- specifica un cammino a partire dalla *current working directory*
- es. pippo/pluto
- es. ../guest/foo.txt
  (riconosciuti da Windows e da Unix)

I nomi speciali "." e ".."

- "." indica la directory corrente
- ".." indica la directory superiore a quella corrente

esempio: "./pippo.txt", equivale a "pippo.txt"

(e "www/index.html" equivale a
"www/../www/../www/././index.html")

Directories

Una directory è una *tabella* che associa il *nome* di un file alla sua locazione sul file system

Cosa deve fare open("/home/matteo/foo.txt", O_RDONLY); ?
0. apri "/", e cerca "home"
1. apri "home" e cerca "matteo"
2. apri "matteo" e cerca "foo.txt"
3. apri "foo.txt"

Ad ogni passo la ricerca può fallire (es. file not found, mancano i permessi)

Per ogni directory sul cammino, devo avere il permesso "x"

Operazioni sulle directory

- Link: "collega" un file a una directory
  Unix: comando ln(1), syscall link(2)
- Unlink: "scollega" un file da una directory
  Unix: comando rm(1), syscall unlink(2)

  In Unix, un file (regolare) può apparire in più di una directory (hard link)

  ⇒ un file è veramente cancellato se sono rimossi *tutti* i link

- Attraversamento: quando accedo a un file che sta "sotto" la directory
- Creazione: in Unix, comando mkdir(1), syscall mkdir(2)
- Rimozione: comando rmdir(1), syscall rmdir(2)

Lettura dei contenuti di una directory

In Unix, una directory è un file

(in Unix, *tutto* è un file)

Posso (in teoria) esaminare il contenuto di una directory con open(2) e read(2)

Il formato della directory dipende dal tipo di file system

⇒ meglio usare le funzioni di libreria opendir(3), readdir(3), close-dir(3)

Implementazione dei file system

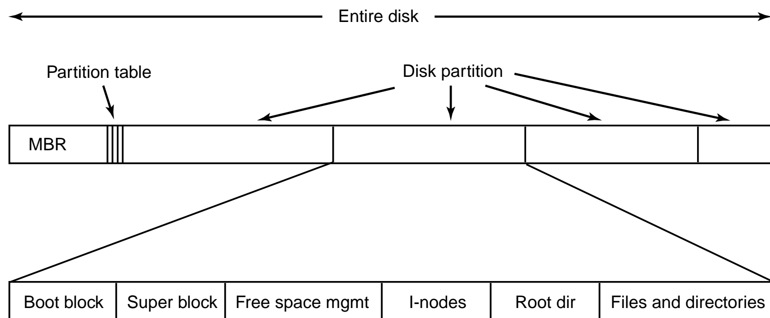
Scopo: organizzare i dischi (collezioni di blocchi) in File Systems (collezioni di file)

Come trovare i blocchi che appartengono a un file X?

Come trovare i blocchi liberi?

Come rappresentare le directory?

Dove conservare gli attributi dei file?



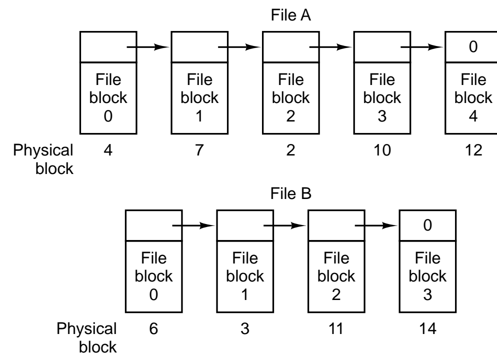
Allocazione contigua

- facile da implementare
- veloce
- richiede file di dimensione prefissata: realistico solo per dispositivi read-only

Implementazione delle directory: tabella che associa il nome al # di blocco iniziale

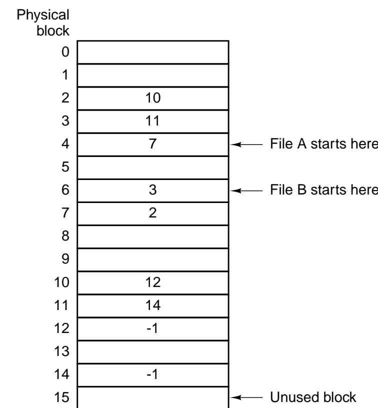
Allocazione in linked list

- ciascun blocco fisico contiene il puntatore al successivo
- molto lenta: devo fare una richiesta di I/O per ogni blocco
- riduce la dimensione dei blocchi logici



Allocazione in linked list con indice (FAT di MS-DOS)

- prendo tutti i puntatori e li metto in un indice
- l'indice resta in RAM ⇒ lettura veloce
- svantaggio: consumo molta RAM



Implementazione delle directory in MS-DOS

Una directory è una tabella che associa il nome al # di blocco iniziale

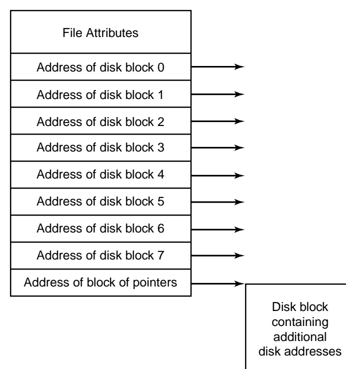
La directory entry contiene anche gli attributi del file

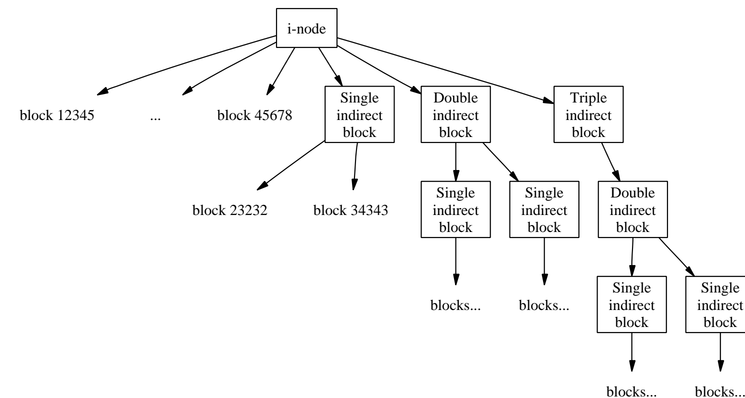
Organizzazione con Inodes (Unix)

- Ciascun file ha associato un inode (index-node)
- Lo inode contiene:
  - attributi
  - gli indirizzi dei primi 10 blocchi fisici
  - l'indirizzo di un single indirect block
  - l'indirizzo di un double e di un triple indirect block

Ogni volume contiene un numero fissato di i-nodes, reperibili in base al numero

Un semplice modello di inode





i-nodes structure in UNIX

Struttura delle directory

Una directory è un file

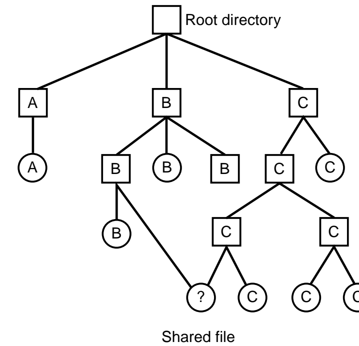
Contiene una lista di directory entries

In Unix ogni entry contiene
- il numero di i-node
- il nome del file

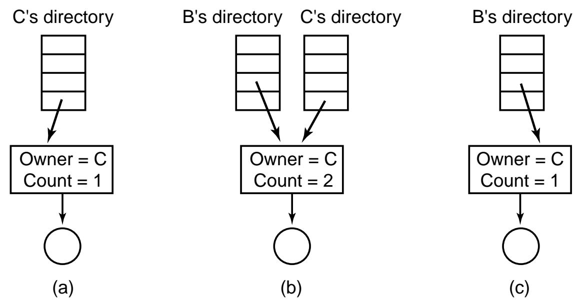
In MS-DOS (FAT)
- il nome del file
- gli attributi
- il numero del primo blocco

Hard link

L'esistenza degli inode permette di implementare gli *hard link*



Implementazione degli hard link



Hard link e soft link

Due (hard) link a uno stesso file sono "indistinguibili"

Per evitare cicli nel FS, le directory possono avere solo 1 hard link

Questa limitazione non vale per i soft link (o symbolic link)

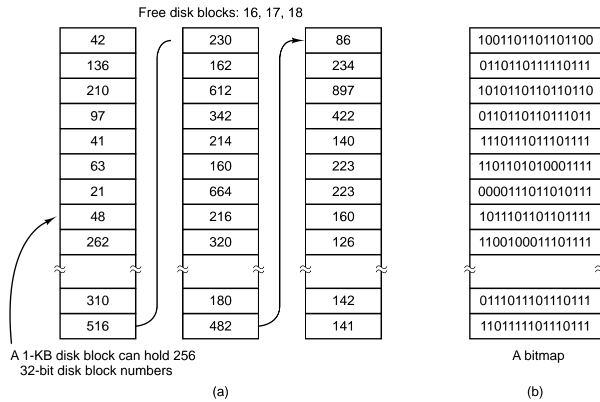
I soft link sono distinguibili (con fstat(2))

Un soft link può essere "pendente"

Un soft link è implementato conservando il pathname del file bersaglio

Gestione dei blocchi liberi

Due strategie:
- free list
- bitmap



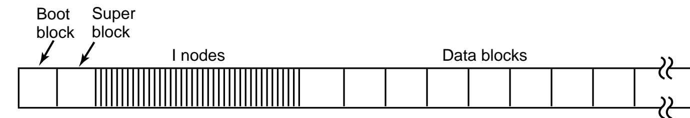
Implementazione del file system di Unix V7

Block 0: usato per il boot

block 1: superblock
- number of inodes, number of blocks, start of free list

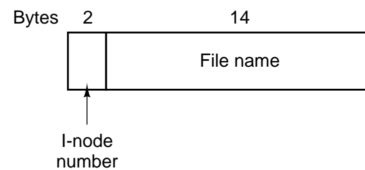
Inodes table
- fixed size

Data blocks

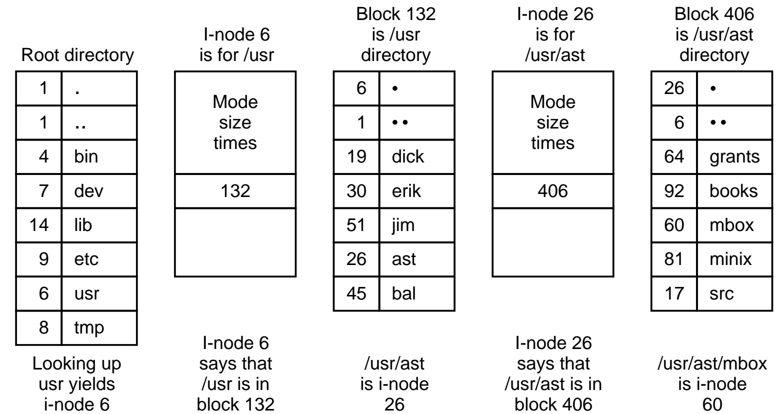


Directory entry in Unix V7

A directory is an unsorted list of entries

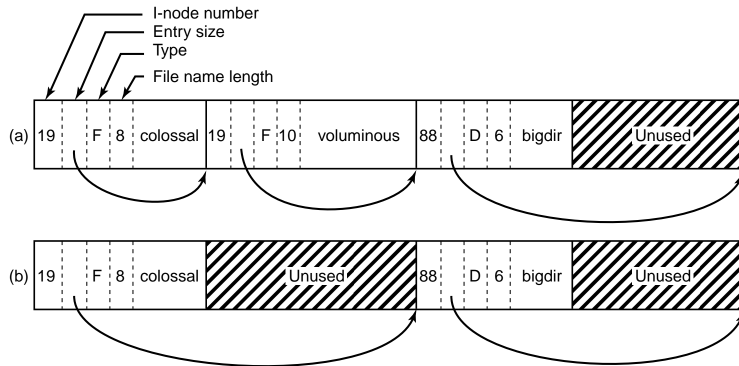


I passi necessari per aprire /usr/ast/mbox



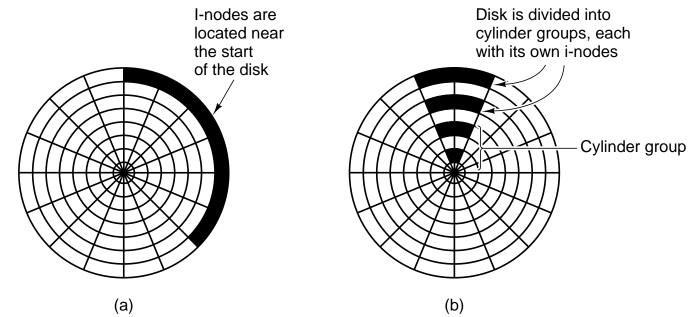
The Berkeley Fast File System (FFS)

File names up to 255 char



The Berkeley Fast File System (FFS) (ii)

Cylinder groups: si cerca di tenere i data block vicino al loro inode



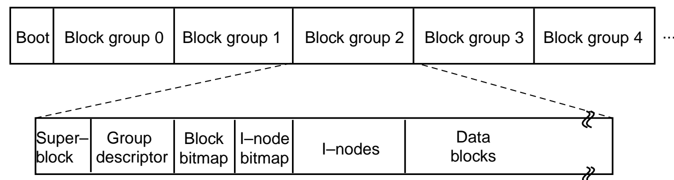
The Linux Ext2 File System

I dischi a geometria virtuale rendono obsoleti i cylinder group

Ext2 divide il disco in block groups contigui, senza tener conto della geometria

Ciascun block group è un FS in miniatura: superblock, inode table, free blocks bitmap, data blocks

Si cerca di allocare blocchi sempre nello stesso block group dell'i-node



Altre ottimizzazioni in Ext2

Quando si deve aggiungere un data block a un file, si cerca di prenderlo consecutivo

Se non è possibile si cerca di prenderlo dallo stesso block group

Un file nuovo prende il primo data block dallo stesso group della sua directory

Le directory sono sparse uniformemente su tutto il FS

Tipi di filesystem

Filesystem speciali
- non corrispondono a veri dispositivi hardware
- proc: fornisce informazioni sullo stato del kernel
- tmpfs: per file temporanei conservati in ram
- shm: fornisce regioni di memoria condivise per IPC

Filesystem basati su dispositivi a blocchi
- ext2: il filesystem tradizionale di Linux
- ext3: la versione “journaled” di ext2
- reiserfs, jfs, xfs: altri filesystem con “journaling”
- vfat: il filesystem di Windows 95

Filesystem di rete
- NFS: tradizionale di Unix
- SMB: di Microsoft, implementato in “Samba”

Montare un filesystem

Ogni filesystem ha una *root directory*

Il filesystem la cui root coincide con la root del sistema è il *root filesystem*

Altri filesystem possono essere *montati* sull’albero delle directory

La directory su cui è montato un FS è detta *mount point*

Dopo la mount, i contenuti precedenti della directory sono nascosti

Montare un filesystem (ii)

mount -t *fstype* [-o options] *device dir*
monta un filesystem specificato da *device* sulla directory *dir*

mount -t vfat /dev/hda2 /win

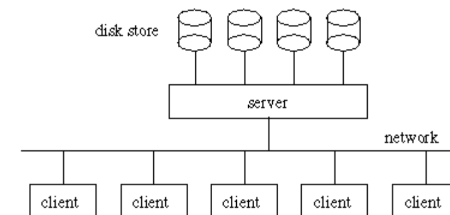
mount -t ext2 -o rw,noatime /dev/hdb1 /mnt/altrodisco

Network File System

Realizzato da Sun nel 1985

La specifica è nel public domain (cf. SMB)

Client-server



NFS (ii)

transport-independent

presume che client e server siano connessi in rete locale veloce

sicurezza:
- il client presenta lo UID e i GID dell'utente al server

⇒ si presume che il client non "bari"
- i dati sono trasmessi in chiaro

⇒ si presume che sia usato in una rete sicura

NFS è stateless (⇒ robust)
- ogni richiesta è autocontenuta
- il server può fare un reboot senza causare errori alle app. client
- la performance non dipende dal # di client, ma dal traffico

Installare NFS

Server:

```
# /etc/exports: NFS file systems being exported.  See exports(5).
/    192.168.1.9(rw)
/pub (ro)
```

Client:

```
# mount 192.168.1.2:/ /mnt/nfs
```


Il protocollo NFS

request	**action**	**idempotent**
GETATTR	get file attributes	yes
SETATTR	set file attributes	yes
LOOKUP	look up file name	yes
READLINK	read from symbolic link	yes
READ	read from file	yes
WRITE	write to file	yes
CREATE	create file	yes
REMOVE	remove file	no
RENAME	rename file	no
LINK	create link to file	no
SYMLINK	create symbolic link	yes
MKDIR	create directory	no
RMDIR	remove directory	no
READDIR	read from directory	yes
STATFS	get filesystem attributes	yes

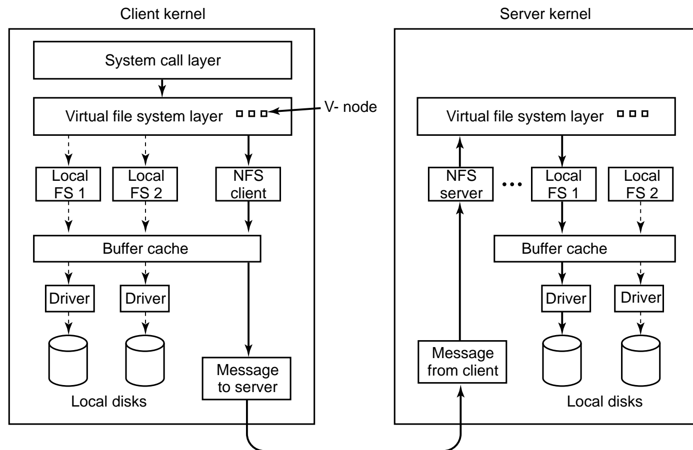
Il protocollo NFS (ii)

LOOKUP (*pathname*):
- il server verifica se l'utente ha diritto di vedere il file
- il server restituisce una *file handle*
- la handle identifica il file per tutte le richieste successive
- la handle è composta da:

filesystem id + inode # + file creation date

La handle è *time stable*: è valida fintantoché il file esiste

The VFS layer structure



Virtual File System

Un sistema Unix deve poter fare I/O in maniera trasparente su tutti i diversi tipi di filesystem

La soluzione è nell'object-orientation

In Linux ogni filesystem deve implementare diverse *classi* di oggetti:

- superblock
- vnode
- file object
- dentry

VFS si basa su un *common file model*

Il CFM di VFS è basato sul FS di Unix

Virtual File System (ii)

superblock object

- contiene informazioni su un FS montato:
  - opzioni di mount(2)
  - block size
  - max size for files

vnode object

- corrisponde a un i-node dei filesystem di Unix
- contiene tutti i dati dell'i-node
- per i FS che non hanno gli inode (FAT) il vnode esiste solo in RAM

file object

- rappresenta l'interazione fra un processo e un file
- contiene il current file offset

Virtual File System (iii)

Per il VFS una directory:

0. è un file
1. contiene una lista di associazioni nome-vnode

dentry object

- rappresenta un associazione fra nome e vnode
- vengono creati durante il "pathname lookup"
- sono conservate in una cache