## Prossime lezioni

- mercoledì 28, 9:00–13:00 Aula 7 Morselli recupero festività
- giovedì 29, non c'è lezione causa Giro D'Italia

## La pipe ("tubo" in inglese)

Meccanismo di comunicazione fra processi

Il più vecchio, condiviso da tutti i sistemi Unix

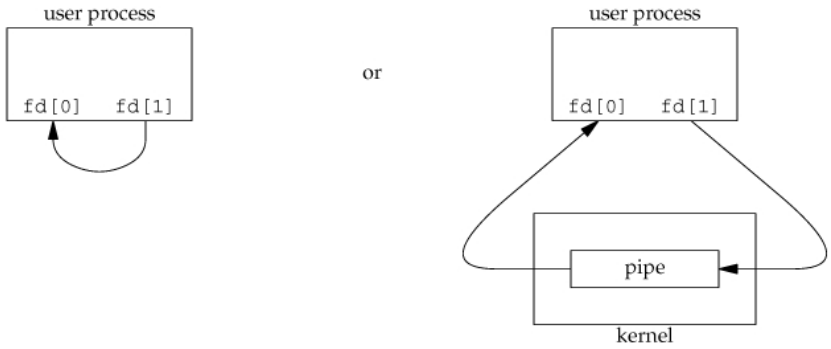
Ha due limitazioni:
- half-duplex: i dati scorrono in una sola direzione
- può essere usato solo fra processi che condividono un antenato

int pipe(int fd[2]);

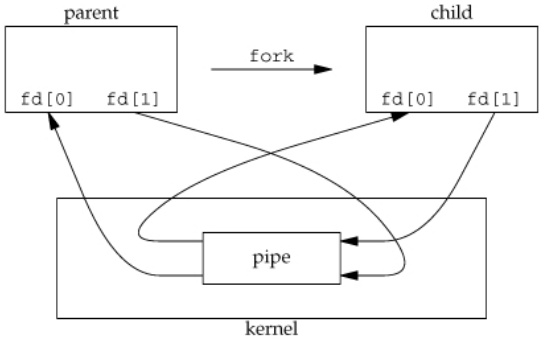
Restituisce 0 se ha successo, -1 per errore

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

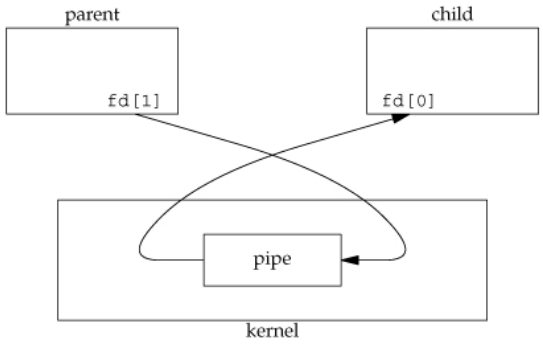
## Una pipe in un singolo processo




## Dopo una fork



## Dopo aver chiuso fd che non servono




## 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]

```
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

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

## Esercizio di laboratorio

Come si implementano le seguenti redirezioni?

- `echo ciao > /tmp/foo`
- `echo ciao >> /tmp/foo`
- `sort < /etc/passwd >> /tmp/foo`
- `sort < /etc/passwd >> /tmp/foo 2>> /tmp/bar`
- `sort < /etc/passwd | grep root`
- `sort < /etc/passwd | grep root | tr o 0`

Scrivere programmi C che implementano il comando usando le chiamate di sistema di Unix

```
% ls | less

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'', ...);
 .                   |                 |
 |<------------------<----------------+
 |
```

## Implementazione dei file system

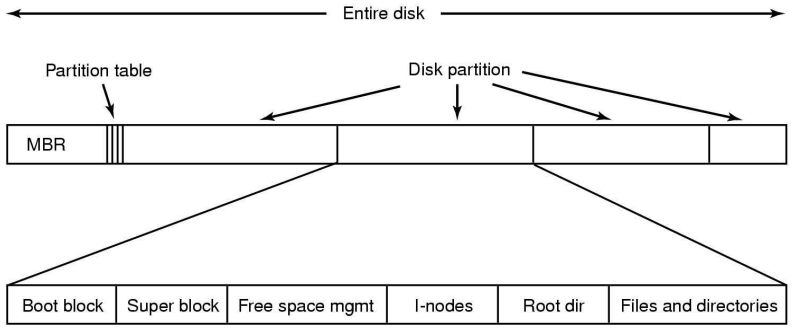
Scopo: organizzare i dischi (collezioni di blocchi) in File Systems (collezioni di file)

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?



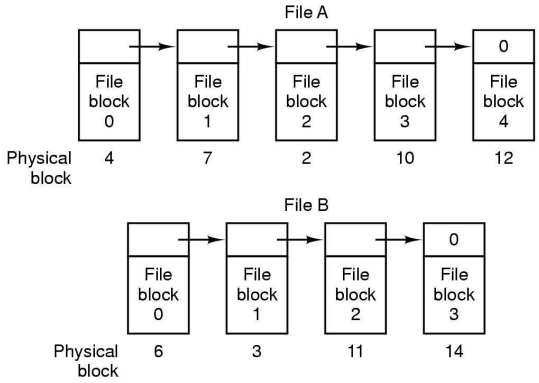

## Allocazione contigua

- 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

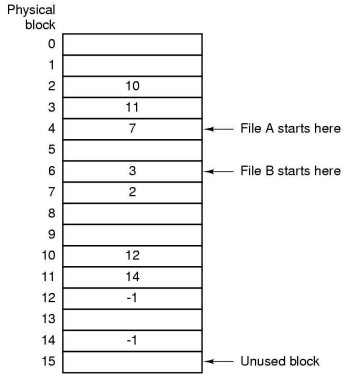
## Allocazione in linked list

- 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)

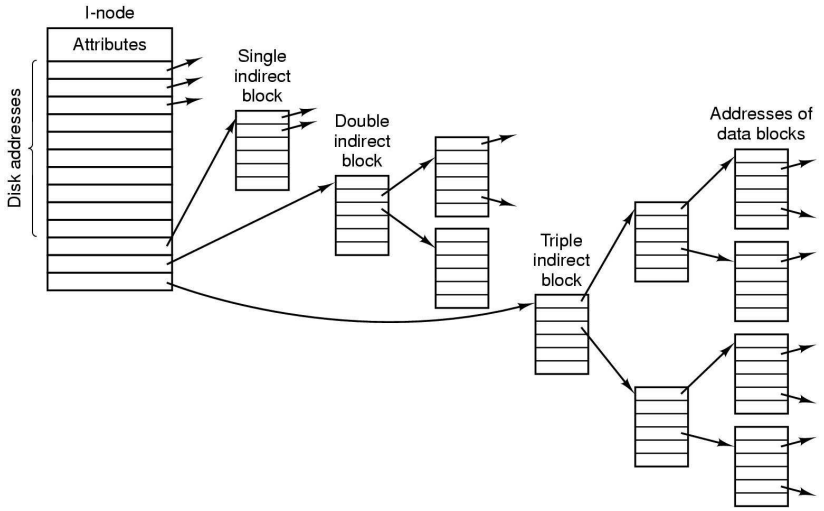
- prendo tutti i puntatori e li metto in un indice
- l'indice resta in RAM ⇒ lettura veloce
- svantaggio: consumo molta RAM




## 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



## Struttura delle directory

Una directory è un file

Contiene una lista di directory entries

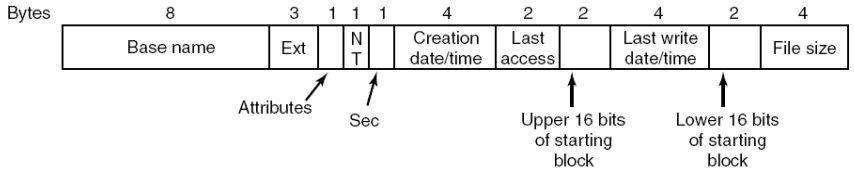
In Unix ogni entry contiene
- il numero di i-node
- il nome del file

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

## Implementazione delle directory in FAT

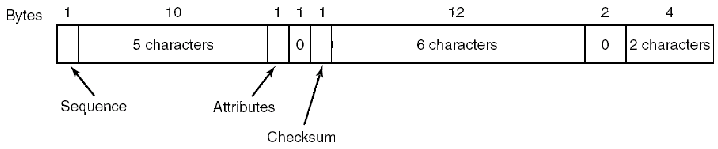
Una directory è una tabella che associa il nome al # di blocco iniziale

La directory entry contiene anche gli attributi del file



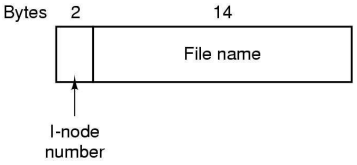

## Implementazione dei nomi lunghi

Il campo "attributes" contiene 0x0F per rendere queste entries "invisibili" ai sistemi pre-Win98

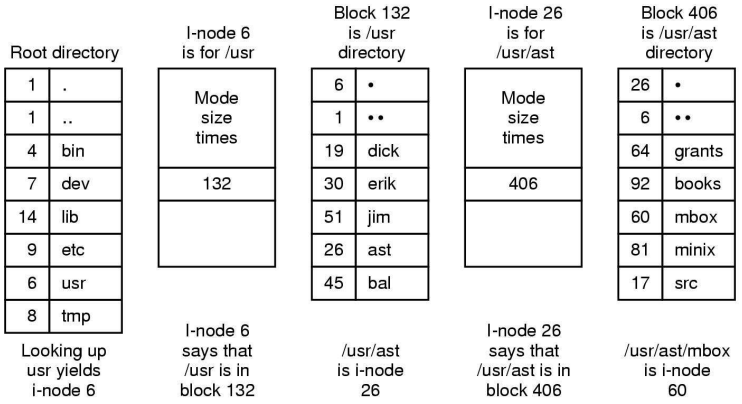



## Directory entry in Unix V7

A directory is an unsorted list of entries

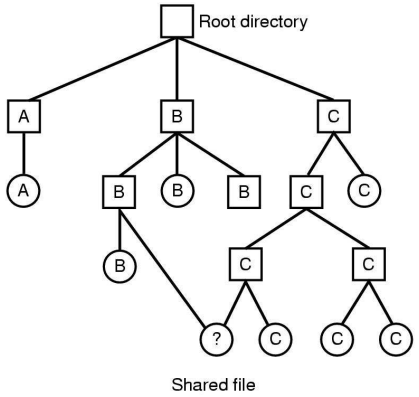


## I passi necessari per aprire /usr/ast/mbox

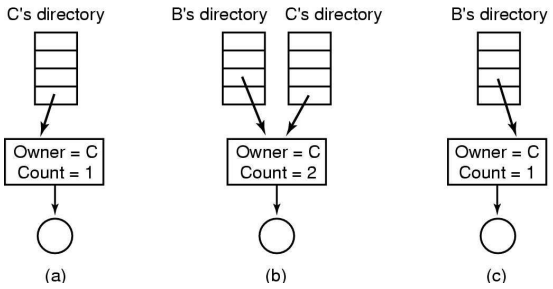



## Hard link

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

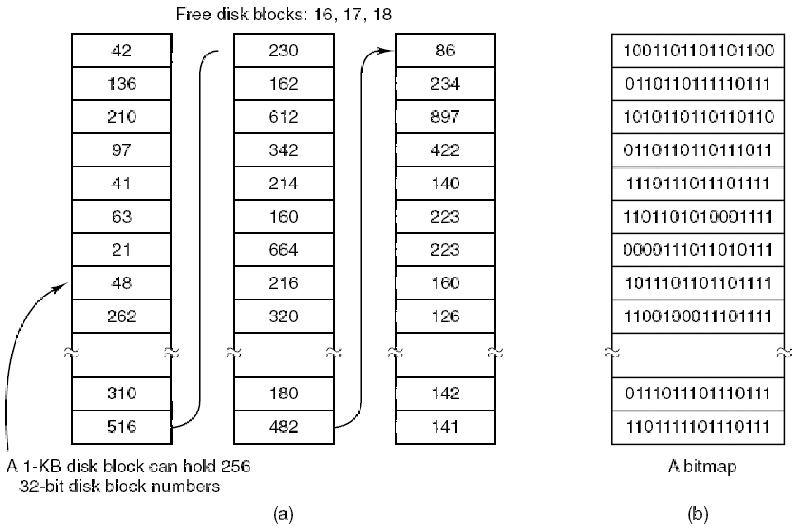
Questa limitazione non vale per i soft link (o symbolic link)

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 con *free list* o *bitmap*




## Implementazione del file system di Unix V7

Block 0: usato per il boot

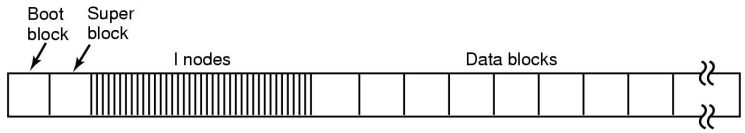
block 1: superblock

- number of inodes, number of blocks, start of free list

Inodes table

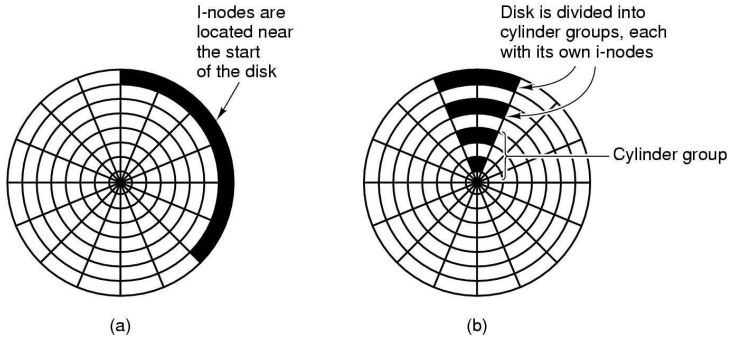
- fixed size

Data blocks

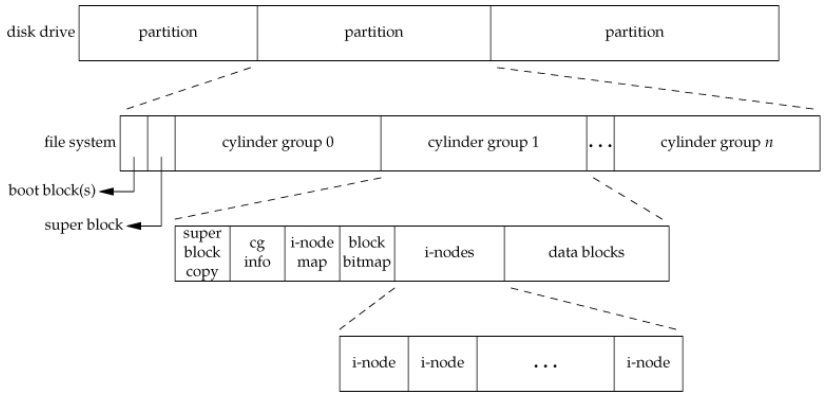



## The Berkeley Fast File System (FFS)

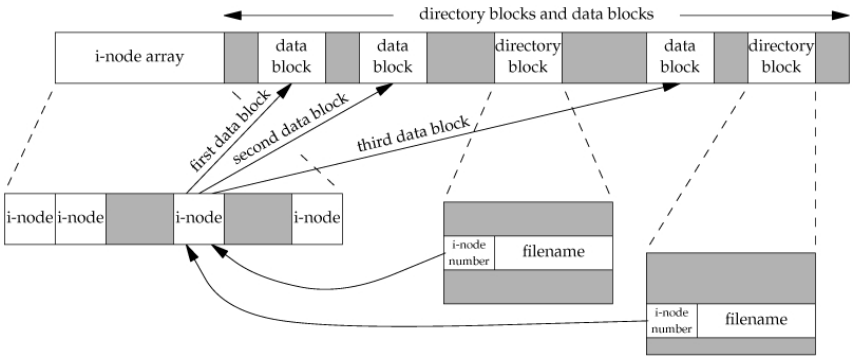
Cylinder groups: si cerca di tenere i data block vicino al loro inode




## The Berkeley Fast File System (FFS)

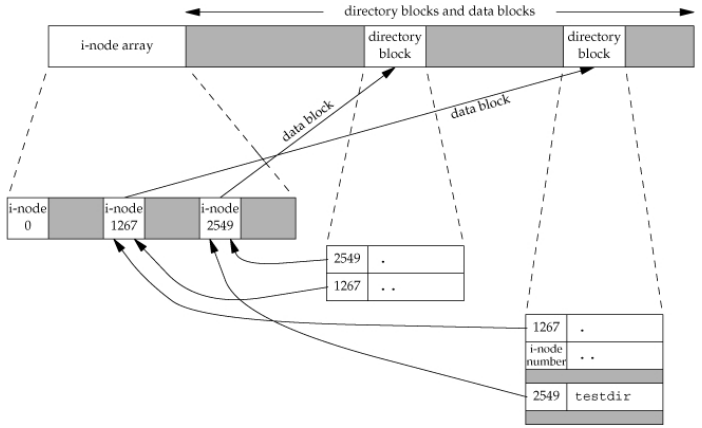


## The Berkeley Fast File System (FFS)



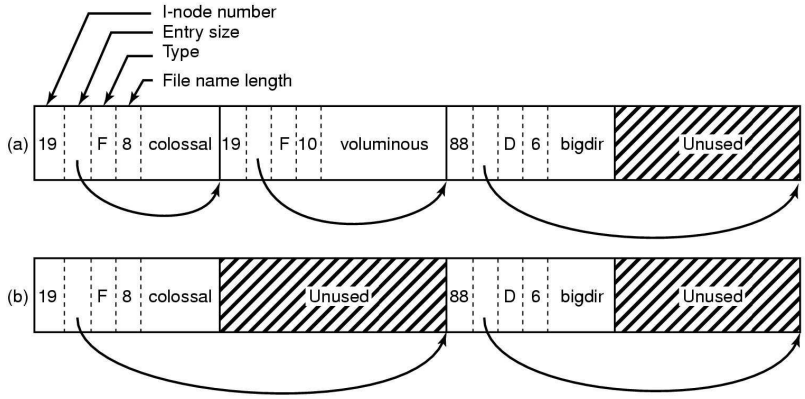

## The Berkeley Fast File System (FFS)

Dopo avere creato la directory "testdir"




## The Berkeley Fast File System (FFS)

File names up to 255 char



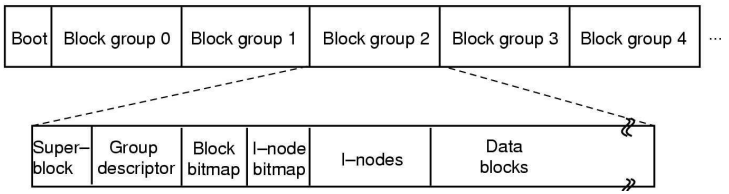

## The Linux Ext2 File System

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'inode



## Altre ottimizzazioni in Ext2

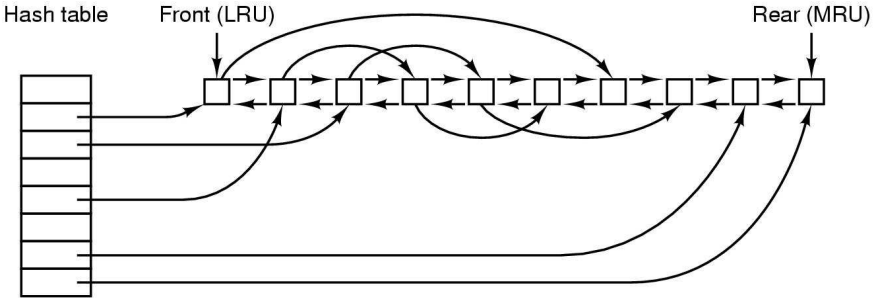
Quando si deve aggiungere un data block a un file, si cerca di prenderlo consecutivo

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

## Buffer cache




(a) catene di collisioni
(b) ordine LRU

```
SYNC(8)                   BSD System Manager's Manual                  SYNC(8)

NAME
  sync - force completion of pending disk writes (flush cache)

SYNOPSIS
  sync

DESCRIPTION
  Sync can be called to insure that all disk writes have been completed
  before the processor is halted in a way not suitably done by reboot(8) or
  halt(8).  Generally, it is preferable to use reboot or halt to shut down
  the system, as they may perform additional actions such as resynchroniz-
  ing the hardware clock and flushing internal caches before performing a
  final sync.

  Sync utilizes the sync(2) function call.
```

```
NAME
   fsync - synchronize a file's in-core state with that on disk

SYNOPSIS
   #include <unistd.h>

   int
   fsync(int fd);

DESCRIPTION
   Fsync() causes all modified data and attributes of fd to be moved to a
   permanent storage device.  This normally results in all in-core modified
   copies of buffers for the associated file to be written to a disk.

   Note that while fsync() will flush all data from the host to the drive
   (i.e. the permanent storage device), the drive itself may not physi-
   cally write the data to the platters for quite some time and it may be
   written in an out-of-order sequence.
```

## open

```
NAME open, creat - open and possibly create a file or device

 ...

 O_SYNC The file is opened for synchronous I/O. Any write()s on the
   resulting file descriptor will block the callzing process until the data
   has been physically written to the underlying hardware. But see
   RESTRICTIONS below.

RESTRICTIONS
  There are many infelicities in the protocol underlying NFS, affecting
  amongst others O_SYNC and O_NDELAY.
```

## Read ahead

Comporamento sequenziale?

Read ahead

## 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

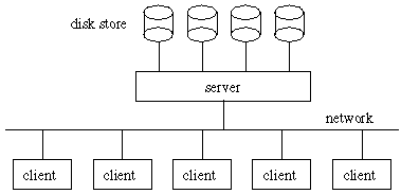
mount -t ext2 -o rw,noatime /dev/hdb1 /mnt/altrodisco

## 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)

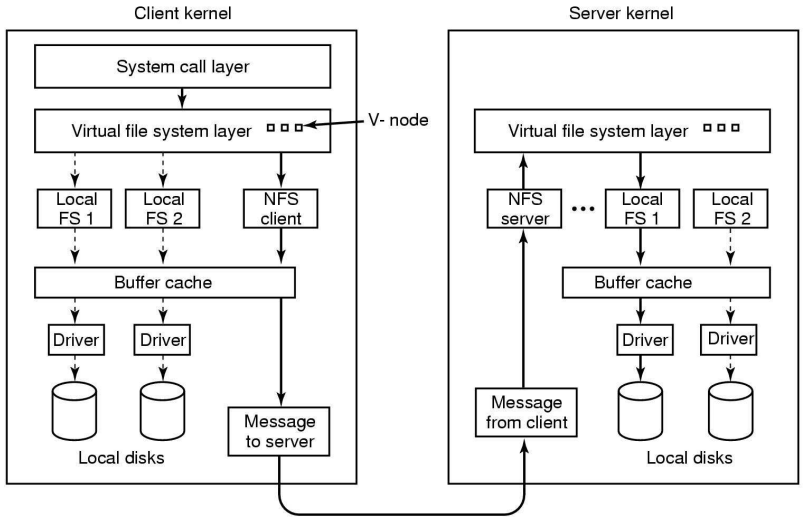
LOOKUP (*pathname*):

- 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

## Una lezione sulle interfacce remote

Quando può fallire una write(2) in locale?

- disco pieno
- crash del disco

Eventi *rari* che si concludono in genere con un crash dell'applicazione

Quando può fallire una write(2) in NFS? spesso e volentieri

## Fallimenti parziali

Una write(2) in locale *aut* riesce *aut* fallisce

Una write(2) in remoto può *non rispondere* e non so se:

- l'operazione non è stata eseguita, perché il comando non è arrivato
- l'operazione è stata eseguita, ma la rete è lenta a rispondere
- l'operazione è stata eseguita, ma la risposta è andata perduta

## E il problema è che ...

Il risultato della write(2) viene generalmente *ignorato* dalle applicazioni

$\Rightarrow$ se ignoro fallimenti intermittenti nella write(2), rischio di corrompere i miei file

*Hard mount* è la regola: se una write non risponde, viene ritentata all'infinito

e il client resta bloccato

$\Rightarrow$ un server cade, e centinaia di workstation si piantano