Blocchi di un file system

Un file deve essere composto da un numero intero di settori
(la testina legge almeno un settore).

I settori di un file non possono essere contigui
(tutti sulla stessa traccia e su tracce dello stesso cilindro)
o far crescere un file sarebbe molto complicato.

Quindi, se un file ha bisogno di spazio,
il file system gli dà un ***blocco*** *di* ***n*** settori contigui,
ma non garantisce che i vari blocchi del file siano contigui.

Ogni blocco ha un ***indirizzo*** (p.es. 0,1,2,...) che il S.O. sa tradurre nell'indirizzo del settore iniziale sul disco.

Quanto dovrebbe valere ***n***?

- Blocchi grossi danno maggiore velocità: leggere 200 settori è più veloce se sono tutti nello stesso blocco, quindi contigui.

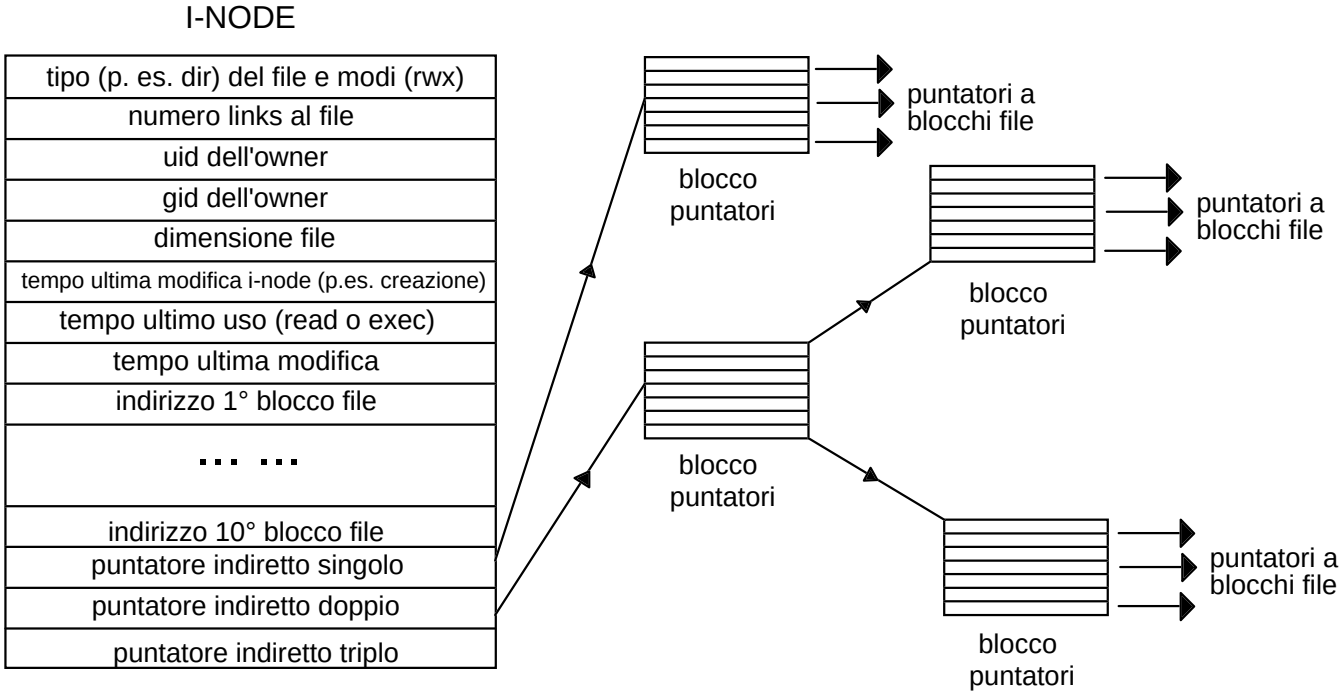
  Inoltre, sono richiesti meno bit per indirizzare blocchi grossi
- Blocchi piccoli sprecano meno spazio (frammentazione interna)
- La scelta comune su UNIX è avere blocchi di 0.5, 1 o 2 K,
  cioè, con settori di 0.5 K, blocchi di 1, 2 o 4 settori.

  $\rightarrow$ 16 bit=2B indirizzano 64K blocchi (64M se blocco = 2 settori)

Il file system di UNIX

In UNIX, ogni file ha un descrittore che si chiama ***i-node***.

Un i-node per un file ha questa struttura:



UNIX spesso ha indirizzi di blocchi a 32 bit (4 byte) e blocchi di 1K.
$\rightarrow$ blocco può contenere 256 indirizzi di (o ***puntatori*** a) blocchi.

- indirizzi dei primi 10 blocchi (10K) del file sono nell'*i-node*
- *puntatore indiretto singolo* = indirizzo di blocco contenente gli indirizzi di al più 256 blocchi di un file (tot. 266 blocchi)
- *puntatore indiretto doppio* = indirizzo di blocco contenente gli indirizzi di 256 blocchi contenenti ciascuno gli indirizzi di 256 blocchi di un file (tot. $266+256^2$=65802blocchi=64Mbyte)
- *puntatore indiretto triplo* consente file di 128 Gigabytes! Per file più grossi si può aumentare la grandezza del blocco.

Tempi di accesso: max 3 accessi per trovare qualsiasi blocco

Struttura della directory in UNIX

Per localizzare un file, occorre quindi il suo i-node. Come si trova?

Ogni directory di UNIX è un file che contiene un “elemento” di 16 byte per ogni file, p. es. `recipes`, contenuto nella directory.
(UNIX non permette di modificare direttamente quest’informazione)

Dei 16 byte di un elemento di directory:

- 14 sono il nome semplice p.es. recipes.
- i primi 2 sono lo ***i-number*** (numero per trovare lo i-node). Quindi in tutto si possono avere 64K i-nodes. P. es.

```
$ od -cb .
0000000    4   ;   .  \0  \0  \0  \0  \0  \0  \0  \0
          064 073 056 000 000 000 000 000 000 000 000
0000020   273   (   .   .  \0  \0  \0  \0  \0  \0  \0
          273 050 056 056 000 000 000 000 000 000 000
0000040   252   ;   r   e   c   i   p   e   s  \0  \0
          252 073 162 145 143 151 160 145 163 000 000
```

Problema 1: trovare lo i-node del file `/usr/jim/recipes`

- lo i-node della root sta in un posto prefissato nel disco.
- nella root, la entry `usr` dà lo i-node di `usr`
- ora si può esaminare `usr` trovando lo i-node di `jim`
- allo stesso modo si trova lo i-node desiderato di `recipes`

Problema 2: trovare lo i-node del file `../work`

- lo i-node della dir corrente è noto
- nella dir si trova lo i-node di `..` (nell’esempio sopra `273 050`)
- si procede come prima

I-nodes e `ls`

La nozione di i-node contribuisce a spiegare le opzioni di `ls`.

- `ls -i` mostra lo i-number per ogni file listato
- `ls -l` mostra il tempo di modifica (scrittura)
- `ls -lc` mostra il tempo di modifica dell'i-node
- `ls -lu` mostra il tempo dell'ultimo uso (lettura o esecuzione)

Inizialmente gli effetti delle opzioni coincidono:

```
$ ls -l junk
ls: file junk not found
$ date
Tue Sep 27 08:08:14 EST 1992
date >junk
$ ls -l junk
-rw-rw-rw- 1 utente1         29 Sep 27 08:08 junk
$ ls -lc junk
-rw-rw-rw- 1 utente1         29 Sep 27 08:08 junk
$ ls -lu junk
-rw-rw-rw- 1 utente1         29 Sep 27 08:08 junk
```

Alle 12:07, `junk` cambia, `ls -l` e `ls -lc` danno la nuova ora

```
$ date
Tue Sep 27 12:07:24 EST 1992
date >junk
$ ls -l junk
-rw-rw-rw- 1 utente1         29 Sep 27 12:07 junk
$ ls -lc junk
-rw-rw-rw- 1 utente1         29 Sep 27 12:07 junk
```

Ma `ls -lu` mostra ancora la vecchia:

```
$ ls -lu junk
rw-rw-rw- 1 utente1         29 Sep 27 08:08 junk
```

Se poi si cambiano i permessi per `junk`, solo `ls -lc` varia

```
$ chmod o-rw junk
$ ls -lc junk
rw-rw---- 1 utente1         29 Sep 27 12:11 junk
```

Link sui file

Un file può avere 2 nomi (albero directory → *grafo diretto aciclico*).

Comando: `ln` *src dest*

Per dare al file `/bin/ed` anche il nome `/user/bin/edt`:

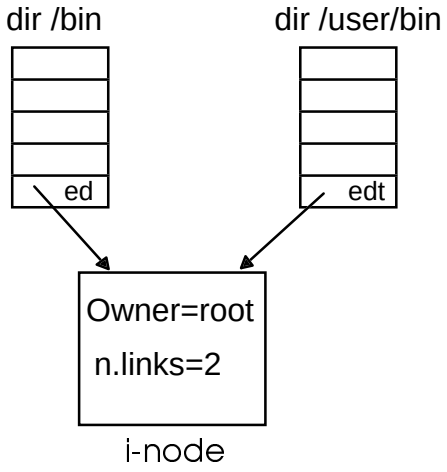
```
# ln /bin/ed /user/bin/edt
```

- in `/user/bin` compare un elemento di 16 byte, con nome `edt` e i-number uguale a quello di `/bin/ed`
- nell'i-node di `/bin/ed` (e ora di `/user/bin/edt`), n.link++

Infatti:

```
# ls -li /user/bin/edt
15768 -rwxrwxrwx 2 root      27920 Sep  8 1988 /bin/ed
15768 -rwxrwxrwx 2 root      27920 Sep  8 1988 /user/bin/edt
```

I due file sono dunque del tutto indistinguibili (stesso i-node):



Problema 1. `rm /bin/ed` dovrebbe cancellare anche `/user/bin/edt` e l'i-node; ma non c'è sufficiente informazione per trovare `/user/bin/edt`; quindi:
`rm` decrementa n. links in i-node, ma non può eliminare i-node e `/usr/bin/edt`

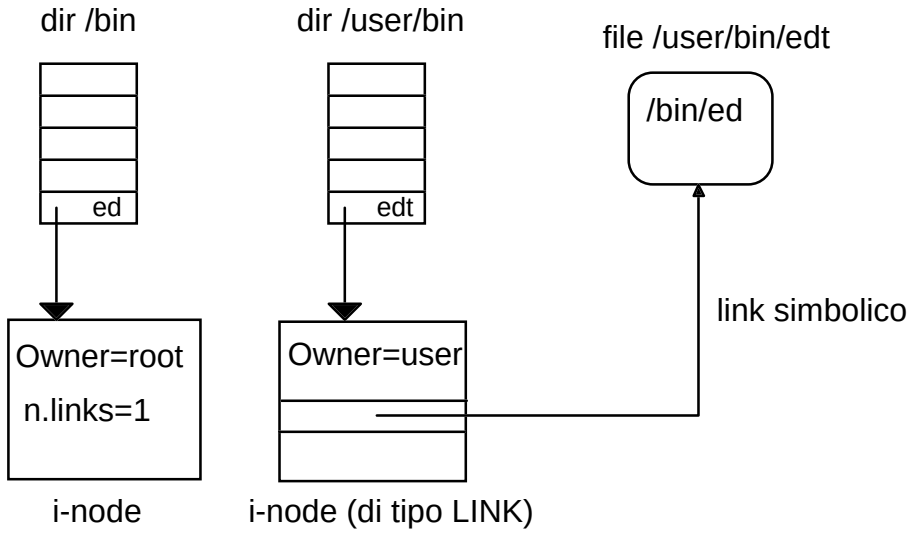
Problema 2. Dopo la cancellazione di `/bin/ed`, root è ancora owner di i-node; se c'è accounting paga per l'uso dei blocchi, ma non può né vuole usarli:
solo l'owner di `/user` continua a usarli, attraverso `/user/bin/edt`

Link simbolici

L'alternativa al link ordinario è il link simbolico: `ln -s` *src dest*

```
$ ln -s /bin/ed /user/bin/edt
```

- crea un file `/user/bin/edt` contenente i caratteri `/bin/ed`
- `/user/bin/edt` è un nuovo file con i-node nuovo di tipo *link*
- FS Unix usa il nome memorizzato in `/user/bin/edt` (link) per trovare l'originale `/bin/ed`
- ma per le applicazioni (p.es. shell) `/user/bin/edt` è un file a pieno titolo, di contenuto coincidente con quello di `/bin/ed`



Vantaggi: la cancellazione del link non crea problemi,
la cancellazione dell'originale nemmeno,
il nome contenuto nel link può essere anche di rete

Svantaggi: + spazio richiesto (i-node e blocco per il link);
+ tempo necessario per gli accessi.

Svantaggio di link simbolici e non: programmi ricorsivi come du rischiano di essere ingannati (incontrano più volte lo stesso file nella loro visita ricorsiva dell'albero).

Protezione in UNIX

Dominio di protezione di un processo: insieme di risorse a cui esso ha accesso

In Unix ogni utente

- è individuato da uno *user-id* o *uid*
- appartiene a un (solo) gruppo, individuato da *group-id* o *gid*

In Unix ogni risorsa:

- è vista come un file (p.es. periferiche)
- ogni file ha un *owner* definito dall'*uid*, che appartiene a un *gid*
- ogni file ha distinti diritti di accesso `rwx` per:
  - utente *uid*
  - utente con uguale *gid*
  - utente con diverso *gid*

A ogni istante un processo gira con specifici *uid* e *gid* che ne definiscono univocamente il dominio di protezione.

I 9 diritti d'accesso a un file sono memorizzati nel suo i-node come *bit* di protezione:

- solo l'owner del file può modificarne l'i-node, quindi in particolare i permessi con `chmod`
- NB: un permesso `w` per "other" consente di scrivere sul file, ma non sull'i-node e quindi non di usare `chmod`
- per la stessa ragione, l'uso di `chmod` non varia il tempo di modifica del file, ma quello dell'i-node

Dominio di protezione e bit setuid

Per un processo *uid* e *gid* variano nel tempo e con essi il dominio di protezione.

Così processi appartenenti a un certo utente possono eseguire comandi privilegiati, p. es. `passwd` o `mkdir`.

In questi casi, il processo che aveva *uid* corrispondente p. es. a `utente1` assume uid e gid di root.

Il meccanismo che assicura ciò è il ***setuid bit*** dei file eseguibili.
Si tratta di un campo dell'i-node del file eseguibile.
Se il bit è a `1`, `ls -l` mostra un `s` al posto del primo `x`:

```
$ ls -l /bin/passwd
-rwsr-xr-x 1 root        8759 Nov 28 1992 /bin/passwd
```

Un programma con bit setuid `s`:

- viene eseguito con l'uid dell'owner del programma (qui root)
- non con l'uid del processo che ha richiesto l'esecuzione.

Così ogni processo può modificare file importanti, ma solo sotto il controllo di un programma *setuid* "autorizzato" a farlo.

Altro esempio: gioco `pacman` eseguito con uid di `utente1`

Alla fine `pacman` deve scrivere il risultato su un file `hiscore` di cui è owner `games` con `rw-r--r--`

`pacman` non può ma invoca il programma memorizzato nel file `wrthisc` con owner `games` e bit di protezione `rwsr-xr-x`

Permessi e directory

I permessi sulla directory `.` si possono vedere con:

```
$ ls -ld .                              ls -l mostra file e dir contenuti in .
drwxrwxr-x 1 user1          75 Nov 25 08:23 .
```

- *logicamente*: directory = contenitore di file/directory
- *concretamente*: directory = file contenente tabella con una entry (nome,i-node) per ogni file/dir logicamente contenuto
- `75` è la grandezza della directory come file-tabella
- `w` permette di modificare la dir `.` in quanto file-tabella; è quindi necessario per creare e cancellare file (logicamente) nella dir

  NB: per cancellare `/...d/f` basta `w` per `d` (non serve per `f`), ma (vedi sotto) serve anche `x` per tutte le dir in `…/d`
- `r` permette di leggere la directory in quanto file-tabella
- `x`, per una directory, indica permesso di *accesso* alla directory, cioè di uso (esecuzione, lettura …) dei file nella directory; questo permesso è richiesto *oltre* a quello proprio dei file

Esempi: dir e permessi

Permessi nulli per gp sulla home dir di user1

```
(user1 user1) $ ls -ld . *
drwx------   3 user1    users        1024 Mar 25 17:22 .
-rw-------   1 user1    users           2 Mar 25 17:22 f
```

gp non può eseguire ls

```
(gp gp) $ ls -ld /home/user1 ; ls -l /home/user1
drwx------   3 user1    users        1024 Mar 25 17:22 /home/user1
ls: /home/user1: Permission denied
```

user1 dà il permesso di lettura al gruppo di gp e crea un eseguibile runme:

```
(user1 user1) $ chmod g+r .
(user1 user1) $ echo echo ciao >runme ; chmod ug+rx runme
```

Ora gp può leggere il file-tabella /home/user1 ma non i suoi contenuti, anche se rx

```
(gp gp) $ ls -l /home/user1          viene letta la directory, ma non il contenuto
ls: /home/user1/f: Permission denied     i nomi provengono dalla dir=file tabella
ls: /home/user1/runme: Permission denied
(gp gp) $ /home/user1/runme ; cat /home/user1/runme
bash: /home/user1/runme: Permission denied
cat: /home/user1/runme: Permission denied
```

user1 rende il contenuto della sua dir accessibile:

```
(user1 user1) $ chmod g+x .
```

Ora i contenuti di user1 sono accessibili a gp, purché lo siano individualmente:

```
(gp gp) $ ls -l /home/user1
-rw-------   1 user1    users           2 Mar 25 17:22 f
-rwxr-xr--   1 user1    users          10 Mar 25 17:44 runme
(gp gp) $ /home/user1/runme
ciao
(gp gp) $ cat /home/user1/runme /home/user1/f
echo ciao
cat: /home/user1/f: Permission denied
```

Un permesso x ma non r sulla dir,

```
(user1 user1) $ chmod g-r .
```

lascia i file contenuti della dir accessibili a gp, ma per usarli gp deve conoscerne l'esistenza:

```
(gp gp)$ /home/user1/runme; cat /home/user1/runme; ls /home/user1/runme
ciao
echo ciao
/home/user1/runme
(gp gp) $ ls /home/user1          se gp non sa che runme esiste non può scoprirlo
ls: /home/user1: Permission denied
```

Le device

La directory `/dev` contiene file corrispondenti a dispositivi fisici.
P. es.:

```
$ ls -l /dev
crw--w--w- 1 root     0,  0 Sep 27 23:09 console
crw-r--r-- 1 root     3,  1 Sep 28 23:09 kmem
crw-r--r-- 1 root     3,  0 May 08 23:09 mem
brw-rw-rw- 1 root     1, 64 Sep 02 21:49 mt0
crw-rw-rw- 1 root     3,  2 Sep 28 23:39 null
```

→ si possono trattare i dispositivi fisici in modo ***uniforme***, usando l'***astrazione*** fornita dal concetto di file;

ciò sia a livello di uso che di programmazione

P. es per copiare (in modo rudimentale) un file su nastro:

```
$ cp f /dev/mt0
```

Qui sia l'utente che il programmatore di `cp` possono ignorare che `/dev/mt0` è un nastro: ciò che comporta vedere un nastro come un file è a carico del sistema operativo

Le device sono caratterizzate da un modo b (*block*) o c (*character*) e da due numeri:

- *major* (individua il tipo di dispositivo) e
- *minor* (distingue dispositivi simili)

Le device `mem` e `kmem` sono la memoria fisica e
la memoria del *kernel* (sistema operativo).

`/dev/null` serve a buttare l'output di un programma, per vederne solo i messaggi di errore o il tempo di esecuzione (via `time`)
e non l'output

Device (cont.)

```
$ ls -l /dev
...
brw-r---w- 1 root     2,  0 Sep 02 21:49 rp00
brw--w--w- 1 root     2,  1 Sep 02 21:49 rp01
crw--w--w- 1 utente1  1,  4 Dec 01 12:00 tty4
```

`rp00` e `rp01` sono partizioni dello stesso disco.

Per vedere quale contiene il file system di utente:

```
$ mount
rp01 on /usr
```

dunque `rp00` contiene la root e `rp01` contiene `/usr`.

Al boot time il file system contiene già la root `/`,
ma `/usr` è vuoto; riceverà subito il file system di `rp01` con:

```
$ mount /dev/rp01 /usr
```

eseguita dallo script di startup.

Il `mount` è per lo più trasparente, ma:

- non sono ammessi link attraverso file system montati
- se si esaurisce lo spazio disponibile su una partizione (vederlo con `df`) non lo si può ampliare con `mount`

Per `/dev/tty4` si noti come sia divenuto dell'`utente1` che lo usa.

Tutti possono scrivere su `tty4`, p.es. con

```
$ echo ciao > /dev/tty4
```

(`utente1` può impedirlo con **`mesg n`**)