

File & Directory

Capitolo 4 -- Stevens

Tipi di file

- **Regular file** = dal punto di vista del kernel un file regolare contiene testo oppure è binario
- **Directory file** = contiene nomi e puntatori ad altri file; solo il kernel può scriverci
- **Character special file** = usato per individuare alcuni dispositivi del sistema. Es: /dev/tty (la tastiera)
- **Block special file** = usato per individuare i dischi. Es: /dev/sda1
- **Pipe e FIFO** = usati per la comunicazione tra processi
- **Symbolic link** = un tipo di file che punta ad un altro file
- **Socket** = usato per la comunicazione in rete tra processi

Proprietà dei File

Per ogni file il filesystem mantiene delle informazioni:

- Tipo di file (normale, directory, link, speciale, ...)
- Permessi (lettura, scrittura, esecuzione)
- Tempo ultimo accesso
- Tempo ultima modifica
- Tempo ultimo cambiamento dello stato (permessi)
- User ID del proprietario del file
- Group ID del proprietario del file
- Numero di link che puntano al file
- Altre informazioni

Struttura stat

```
struct stat {  
    mode_t    st_mode;        /* file type & mode (permissions) */  
    ino_t     st_ino;         /* i-node number (serial number) */  
    dev_t     st_dev;         /* device number (filesystem) */  
    dev_t     st_rdev;        /* device number for special files */  
    nlink_t   st_nlink;       /* number of links */  
    uid_t     st_uid;         /* user ID of owner */  
    gid_t     st_gid;         /* group ID of owner */  
    off_t     st_size;        /* size in bytes, for regular files */  
    time_t    st_atime;       /* time of last access */  
    time_t    st_mtime;       /* time of last modification */  
    time_t    st_ctime;       /* time of last file status change */  
    long      st_blksize;     /* best I/O block size */  
    long      st_blocks;      /* number of 512-byte blocks allocated */  
};
```

Permessi di accesso ai File

► **st_mode** nella **struttura stat** include anche 9 bit che regolano i permessi di accesso al file cui esso si riferisce

st_mode mask	Meaning
S_IRUSR S_IWUSR S_IXUSR	user-read user-write user-execute
S_IRGRP S_IWGRP S_IXGRP	group-read group-write group-execute
S_IROTH S_IWOTH S_IXOTH	other-read other-write other-execute

stat, fstat e lstat

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
int stat (const char *pathname, struct stat *buf);
```

```
int fstat (int fd, struct stat *buf);
```

```
int lstat (const char *pathname, struct stat *buf);
```

Descrizione: le informazioni sul file passato come 1° arg. vengono memorizzate nella struttura passata come 2° arg.

Restituiscono: 0 se OK

-1 in caso di errore

Funzioni stat, fstat, lstat

- **stat**: fornisce una struttura di info relative al file del primo argomento
- **fstat**: come prima, ma il file cui si riferisce è già aperto e quindi prende come argomento il file descriptor
- **lstat**: le info ottenute sono relative al link simbolico (e non al file a cui esso si riferisce)

Funzioni stat, fstat, lstat

- per tutte: bisogna fornire un puntatore ad una struttura (di tipo **stat**) che viene poi riempita durante lo svolgimento della funzione.
- Un grande utilizzatore di tali funzioni è il comando shell
`ls -l`

fornisce informazioni circa un file dato come argomento

```
bash> ls -l file.c
-rwxrw-rw-  1 rescigno  14441 Mar 18 16:35 file.c
```


Macro per tipi di file

►► Le macro seguenti sono funzioni booleane che aiutano ad identificare il tipo di un file verificando ciò che è contenuto nel campo **st_mode** della struttura stat del file

Macro	Type of file
S_ISREG ()	regular file
S_ISDIR ()	directory file
S_ISCHR ()	character special file
S_ISBLK ()	block special file
S_ISFIFO ()	pipe or FIFO
S_ISLNK ()	symbolic link (not in POSIX.1 or SVR4)
S_ISSOCK ()	socket (not in POSIX.1 or SVR4)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;    struct stat    buf;

    for (i=1;i<argc;i++){
        printf("%s:", argv[i]);
        if (lstat(argv[i],&buf) <0) {
            printf("lstat error");
            continue;
        }
        if (S_ISREG(buf.st_mode))    printf("regular");
        else if (S_ISDIR(buf.st_mode)) printf("directory");
        else if (S_ISCHR(buf.st_mode)) printf("character special");
        else if (S_ISBLK(buf.st_mode)) printf("block special");
        .....
    }
    exit(0);
}
```

ID dei processi

- Il campo **st_uid** (**st_gid**) della **struttura stat** contiene l'ID dell'utente (gruppo) possessore del file.
- Ogni **processo** ha degli ID associati:
 - **real u/g ID, effective u/g ID, saved set-u/g-ID**
 - Normalmente effective user ID coincide con real user ID

ID dei processi

- **real** : chi siamo realmente
 - presi dal file */etc/passwd* al login time
- **effective** : determina i permessi di accesso ai file
- **saved set** : contengono copie dell'effective quando è eseguito un programma (exec)

Set-User-ID & Set-Group-ID

- quando un programma è eseguito normalmente
`effective=real`
- ...ma si può settare un flag speciale nel campo **st_mode** che fa sì che il processo sia eseguito con `effective=proprietario` (o group) del file eseguibile
- Questi bit possono essere testati usando le costanti `S_ISUID` e `S_ISGID`

Set User-ID

pippo.doc è un file di pippo sul quale solo pippo può scrivere

scrivi è un word-processor di pippo che può essere usato da tutti

pippo può modificare
pippo.doc usando *scrivi*?
SI!

l'utente pluto può modificare
pippo.doc usando *scrivi* di
pippo?

NO! A meno che
scrivi abbia il
set-user-id flag settato



Set User-ID

Questa caratteristica è utilizzata quando cambiamo la nostra password nel sistema

Eseguiamo il comando `/usr/bin/passwd` per modificare il file `/etc/passwd` (di proprietà di root)

esercizi

scrivere un programma che testa se un file ha il flag *set-user-id* settato

- ricorda che i flag set-u/g-ID sono nel campo **st_mode**
- Hint: AND con le costanti S_ISUID e S_ISGID per testarli

inserire un nuovo utente ed implementare esempio precedente

Accesso ai file

- Gli ID dell'owner (user & group) sono proprietà di **file**
 - infatti hanno un campo della struct stat
- Gli effective ID (user & group) sono proprietà del **processo** che utilizza quel file (apri, chiudi, etc.)

Accesso ai file

- Per **aprire** un file (lettura o scrittura) bisogna avere permesso di esecuzione in tutte le directory contenute nel path assoluto del file
- Per **creare** un file bisogna avere permessi di scrittura ed esecuzione nella directory che conterrà il file

Algoritmo di accesso ai file

- eff. uid = 0 --> accesso libero
- eff. uid = owner ID
 - accesso in accordo ai permessi “*User*”
- eff. gid = group ID
 - accesso in accordo ai permessi “*Group*”
- accesso in accordo ai permessi “*Other*”
- Le verifiche sono eseguite esattamente in questo ordine

???

Se il file test ha i seguenti permessi:

_ r _ _ r w _ r w _

chi può modificare il file?

Nuovi file e directory

- quando si creano nuovi file, l'uid è settato come l'effective ID del processo che sta creando il file
- il gid è il group ID della directory nel quale il file è creato oppure il gid del processo

access

```
#include <unistd.h>
```

```
int access (const char *pathname, int mode);
```

Descrizione: verifica se il real ID ha accesso al file 1° argomento nella modalità specificata da *mode*

Restituisce: 0 se OK,
-1 in caso di errore

<i>mode</i>	Description
R_OK	test for read permission
W_OK	test for write permission
X_OK	test for execute permission
F_OK	test for existence of file

```
#include <sys/types.h>
```

```
#include <fcntl.h>
```

```
#include <stdio.h>
```

```
int main(int argc, char *argv[])
```

```
{
```

```
    if (argc != 2)
```

```
        printf("usage: a.out <pathname>");
```

```
    if (access(argv[1], R_OK) < 0)
```

```
        printf("access error for %s", argv[1]);
```

```
    else
```

```
        printf("read access OK\n");
```

```
    if (open(argv[1], O_RDONLY) < 0)
```

```
        printf("open error for %s", argv[1]);
```

```
    else
```

```
        printf("open for reading OK\n");
```

```
    exit(0);
```

```
}
```

```
$ ls -l a.out
$ -rwxrwxr-x 1 rescigno 1234 jan 18 08:48 a.out
$ a.out a.out
read access OK
open for reading OK

$ ls -l prova
$-rw----- 1 basile 1234 jan 18 15:48 prova
a.out prova
access error for prova: Permission denied
open error for prova: Permission denied

$su
# chown basile a.out
# chmod u+s a.out
# ls -l a.out
# -rwsrwxr-x 1 basile 1234 jan 18 08:48 a.out
# exit

$ a.out prova
access error for prova: Permission denied
open for reading OK
```


chmod e fchmod

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
int chmod (const char *pathname, mode_t mode);
```

```
int fchmod (int fd, mode_t mode);
```

Descrizione: cambiano i bit di permesso del file 1° argomento

Restituiscono: 0 se OK, -1 in caso di errore

chmod e fchmod

- Per cambiare i permessi, l'effective uid del processo deve essere uguale all'owner del file, o il processo deve avere i permessi di root
- Il *mode* è specificato come l'OR bit a bit di costanti che rappresentano i vari permessi

Costanti per chmod

<i>mode</i>	Description	
S_ISUID	set-user-ID on execution	4000
S_ISGID	set-group-ID on execution	2000
S_ISVTX	saved-text (sticky bit)	1000
S_IRWXU	read, write, and execute by user (owner)	700
S_IRUSR	read by user (owner)	400
S_IWUSR	write by user (owner)	200
S_IXUSR	execute by user (owner)	100
S_IRWXG	read, write, and execute by group	070
S_IRGRP	read by group	040
S_IWGRP	write by group	020
S_IXGRP	execute by group	010
S_IRWXO	read, write, and execute by other (world)	007
S_IROTH	read by other (world)	004
S_IWOTH	write by other (world)	002
S_IXOTH	execute by other (world)	001

```
#include    <sys/types.h>
#include    <sys/stat.h>

int main(void)
{
    struct stat  statbuf;
    /* turn on set-group-ID and turn off group-execute */
    if (stat("foo", &statbuf) < 0)
        printf("stat error for foo");
    if (chmod("foo", (statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0)
        printf("chmod error for foo");

    /* set absolute mode to "rw-r--r--" */
    if (chmod("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) < 0)
        printf("chmod error for bar");

    exit(0);
}
```



- ...e se invece si vuole settare il group-execute bit?

```
chmod("foo", (statbuf.st_mode | S_IXGRP))
```

sticky bit

- ereditato da versioni "vecchie" di Unix
- se settato, una copia del programma viene salvata nella *swap area* così che la prossima volta che viene lanciato, è caricato in memoria più velocemente
- ora, con la memoria virtuale, non si usa più
- solo il superuser lo può modificare per un file regolare

sticky bit per directory

- se settato, un file in questa directory può essere *removed*, o *renamed* solo se:
 - l'utente ha il permesso di scrittura nella directory, ed
 - è vera almeno una delle seguenti condizioni
 - è proprietario del file
 - è proprietario della directory
 - è *root*
- Esempio: La directory /tmp ha questa caratteristica, infatti in essa qualunque utente ha la possibilità di creare file ed in genere i suoi permessi sono tutti settati a 1, ma non può cancellare o rinominare file di proprietà di altri (anche se si dispongono di tutti gli altri permessi di scrittura necessari).

chown

```
#include <sys/types.h>
```

```
#include <unistd.h>
```

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

```
int fchown (int fd, uid_t owner, gid_t group);
```

```
int lchown (const char *pathname, uid_t owner, gid_t group);
```

Descrizione: cambiano il proprietario ed il gruppo del file 1° argomento e li settano uguale a *owner* e *group*

Restituiscono: 0 se OK,

-1 in caso di errore

dimensione di file

- La dimensione dei files (in bytes) è in **st_size** (nella struttura stat)
- La dimensione del blocco utilizzato nelle operazioni di I/O è contenuto in **st_blksize**
- Il numero di blocchi da 512 byte allocati per il file è contenuto in **st_blocks**

Esercizio 3_3

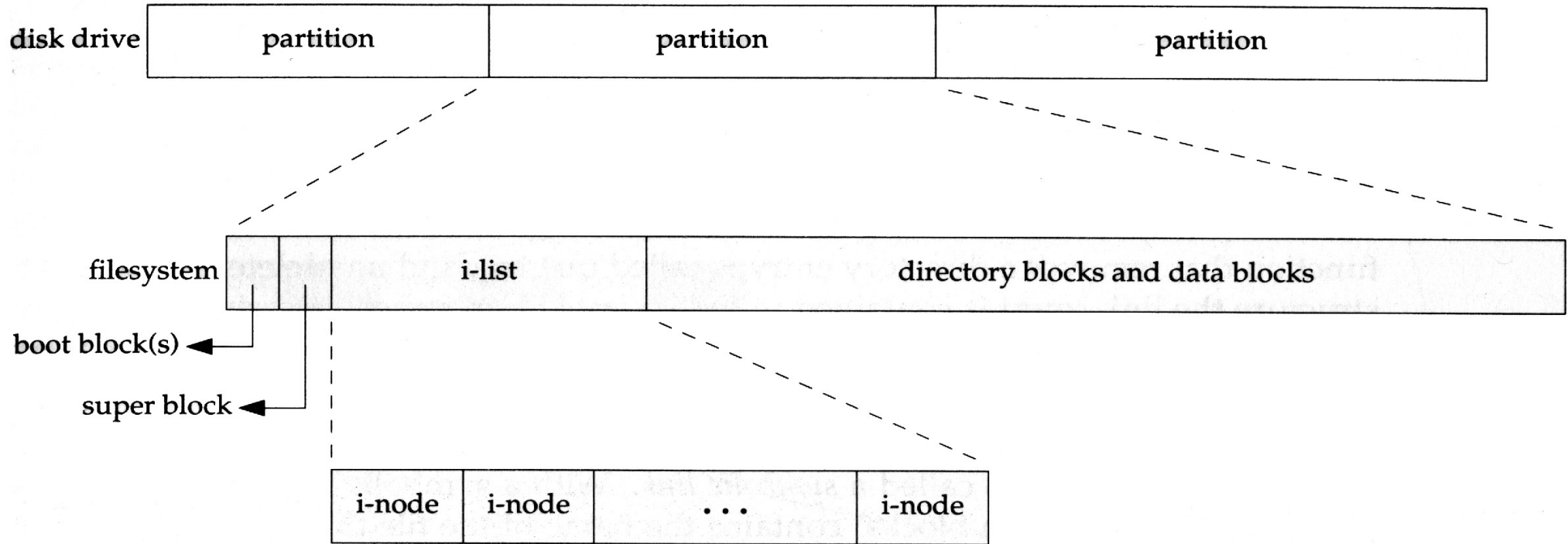
Riscrivere la funzione `access`.

Tale funzione

```
int my_access (const char *path, int mode);
```

avrà come primo parametro un file da testare e
come secondo parametro un numero ottale che
indicherà i permessi da controllare (read, write,
execute o esistenza del file) per il real user.

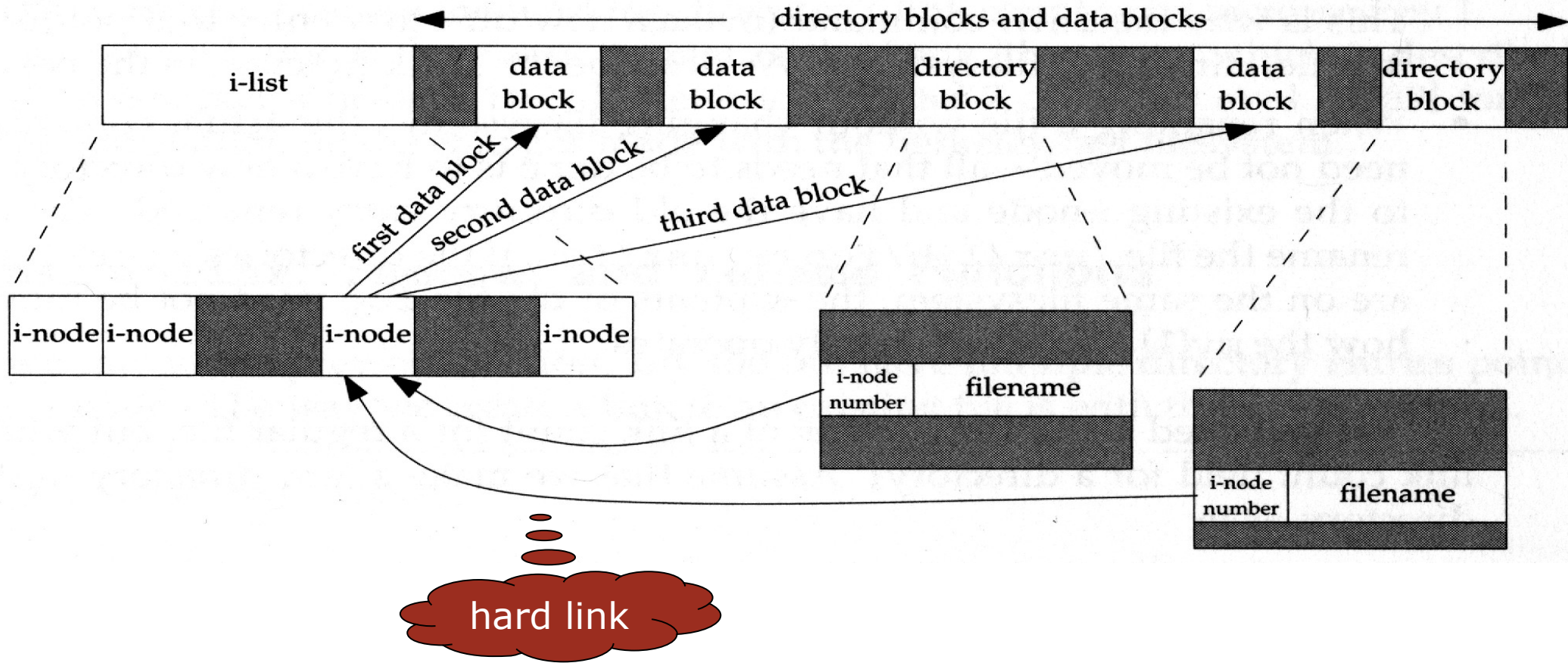
Filesystem



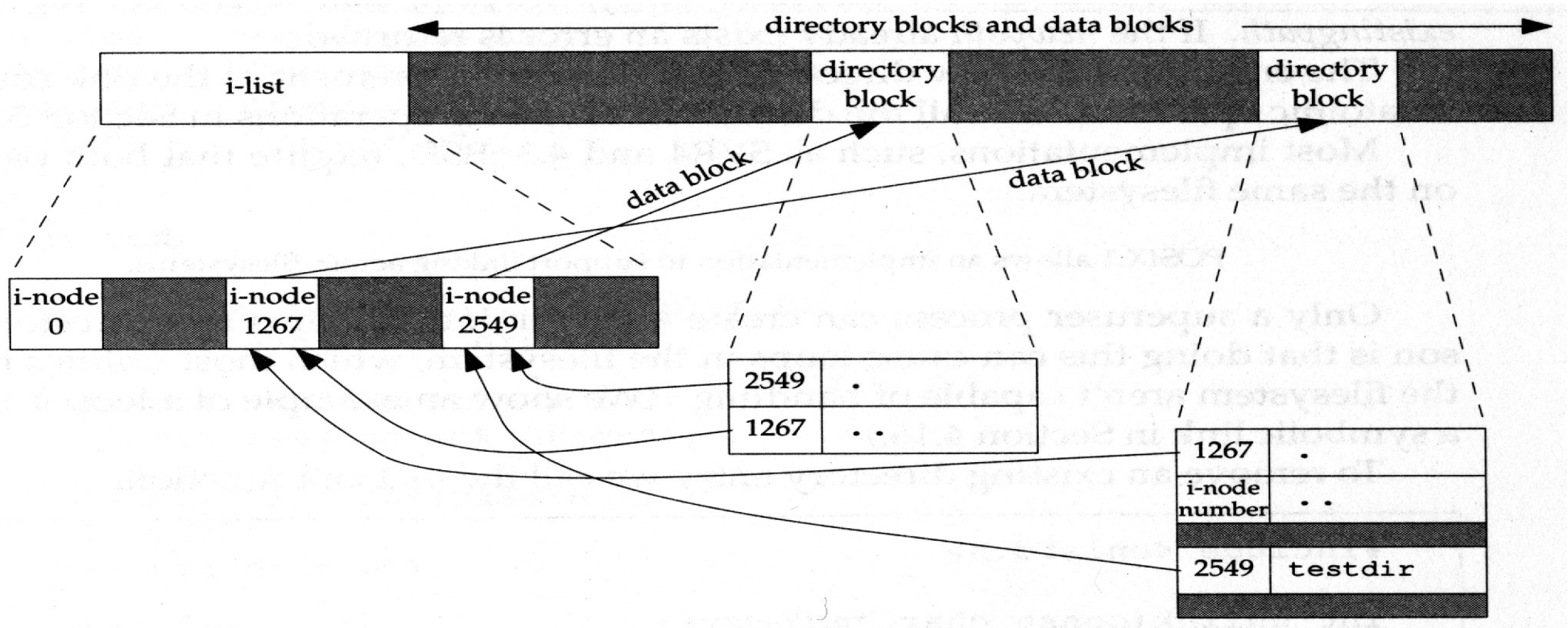
i-node

- i-nodo contiene tutte le info che riguardano il file
 - tipo del file
 - bit di permesso
 - size del file
 - puntatori ai blocchi di dati del file
 -
- directory block: è una lista di record aventi almeno due campi
 - numero dell'i-nodo
 - nome del file

filesystem (i-node)



filesystem: esempio



filesystem: osservazioni

- ogni i-node ha un contatore di link che contiene il numero di *directory entry* che lo puntano
 - solo quando *scende* a zero, allora il file può essere cancellato (i blocchi sono rilasciati...funzione **unlink**)
- il contatore è nella struct stat nel campo **st_nlink**
- questi tipi di link sono detti **hard link**
- non si possono fare hard link tra filesystem differenti

hard link

- un file può avere più di una directory entry che punta al suo i-node, cioè più hard link il cui numero è contenuto in **st_nlink**
- hard link possono essere creati usando la funzione **link**
- solo un processo *super-user* può creare un nuovo link (usando **link**) che punti ad una directory
- **unlink** decrementa il contatore di link
- quando il contatore va a zero, il blocco è rilasciato

link

```
#include <unistd.h>
```

```
int link (const char *path, const char *newpath);
```

Descrizione: crea una nuova directory entry *newpath* che si riferisce a *path*

Restituisce: 0 se OK,

-1 in caso di errore (anche se *newpath* già esiste)

link

- **link** crea un hard link aggiuntivo (con la corrispondente directory entry) ad un file esistente
- **link** crea automaticamente il nuovo link (e quindi la nuova directory entry) ed incrementa anche di uno il contatore dei link **st_nlink** (è una operazione atomica!)
- Il vecchio ed il nuovo link, riferendosi allo stesso i-node, condividono gli stessi diritti di accesso al “file” cui essi si riferiscono

unlink

```
#include <unistd.h>
```

```
int unlink (const char *pathname);
```

Descrizione: rimuove la directory entry specificata da *pathname* e decrementa il contatore dei link del file

Restituisce: 0 se OK,
-1 in caso di errore

unlink

- **unlink** è consentita solo se si ha il permesso di scrittura ed esecuzione nella directory dove è presente la directory entry
- solo il superuser può rimuovere un hard link ad una directory

unlink

Se tutti i link ad un file sono stati rimossi e **nessun processo ha ancora il file aperto**, allora tutte le risorse allocate per il file vengono rimosse e non è più possibile accedere al file

Se però **uno o più processi hanno il file aperto** quando l'ultimo link è stato rimosso, pur essendo il contatore dei link a 0 il file continua ad esistere e sarà rimosso solo quando tutti i riferimenti al file saranno chiusi

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <fcntl.h>
#include    "ourhdr.h"

int main(void)
{
    if (open("tempfile", O_RDWR) < 0)
        err_sys("open error");

    if (unlink("tempfile") < 0)
        err_sys("unlink error");

    printf("file unlinked\n");
    sleep(15);
    printf("done\n");

    exit(0);
}
```

```
$ ls -l tempfile
```

```
-rw-r--r--  stevens    9240990  Jul 31 13:42 tempfile
```

```
$ df /home
```

Filesystem	kbytes	used	avail	capacity	mounted on
/dev/sdoh	282908	181979	72638	71%	/home

```
$ a.out &
```

```
1354
```

```
$ file unlinked
```

```
$ df /home
```

Filesystem	kbytes	used	avail	capacity	mounted on
/dev/sdoh	282908	181979	72638	71%	/home

```
$ ls -l tempfile
```

```
tempfile not found
```

```
$ done
```

```
$ df /home
```

Filesystem	kbytes	used	avail	capacity	mounted on
/dev/sdoh	282908	17293	81678	68%	/home

remove

```
#include <stdio.h>
```

```
int remove (const char *pathname);
```

Descrizione: rimuove il file specificato da *pathname*

Restituisce: 0 se OK,
-1 in caso di errore

rename

```
#include <stdio.h>
```

```
int rename (const char *oldname, const char *newname);
```

Descrizione: assegna un nuovo nome *newname* ad un file od ad una directory data come 1° argomento

Restituisce: 0 se OK,

-1 in caso di errore

Similitudini

- per un file **remove** è identico a **unlink**
- per una directory è identico a **rmdir**

Link simbolici

- hard link non possono *attraversare* file system differenti ed hard link a directory possono essere creati solo dal *superuser*
- i soft link, invece permettono di fare entrambe le cose
- sono dei puntatori indiretti ad un file
 - Es. lib → usr/lib (7 caratteri di dati nel *data block*)
- Quando si usano funzioni che si riferiscono a file (open, read, stat, etc.), si **deve** sapere se *seguono* il link simbolico o no
 - si → ci si riferisce al vero file
 - no → ci si riferisce al link
- vedere figura seguente (es: open si, lstat no)

Link simbolici

Function	Does not follow symbolic link	Follows symbolic link
access		•
chdir		•
chmod		•
chown		•
creat		•
exec		•
lchown	•	
link		•
lstat	•	
open		•
opendir		•
pathconf		•
readlink	•	
remove	•	
rename	•	
stat		•
truncate		•
unlink	•	

Figure 4.17 Treatment of symbolic links by various functions

symlink

```
#include <unistd.h>
```

```
int symlink (const char *path, const char *sympath);
```

Descrizione: crea un link simbolico *sympath* che punta a *path*

Restituisce: 0 se OK

-1 altrimenti

.....link simbolici

- Il file puntato da un link simbolico può non esistere

```
$ ln -s /no/such/file myfile
$ ls myfile
myfile
$ cat myfile
cat: myfile : No such file or directory
$ ls -l myfile
lrwxrwxrwx 1 stevens 13 Dec 6 07:27 myfile->/no/such/file
```

- ▶▶ loop creati con link simbolici possono essere facilmente rimossi con **unlink** (non segue il link simbolico)

readlink

```
#include <unistd.h>
```

```
int readlink (const char *pathname, char *buf, int bufsize);
```

Descrizione: legge il contenuto del link simbolico 1° argomento e lo copia in *buf* la cui taglia è *bufsize*

Restituisce: il numero di byte letti se OK

-1 in caso di errore

readlink

- **readlink** legge il contenuto del link e non del file cui esso si riferisce
- Se la lunghezza del link simbolico è $> \text{bufsize}$ viene restituito errore
- combina insieme le funzioni di open, read e close sul link simbolico

I tempi dei files

- per ciascun file 3 tempi sono gestiti (essi sono presenti nella struttura *stat*)
 - st_atime = la data dell'ultimo accesso al file (read)
 - st_mtime = la data dell'ultima modifica al file (write)
 - st_ctime = la data dell'ultimo cambiamento apportato all'i-node (chmod, chown)
- i tempi di modifica di una directory sono relativi alla creazione o cancellazione dei suoi file non ad operazioni di lettura o scrittura nei suoi file

mkdir

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
#include <unistd.h>
```

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

Descrizione: crea una directory i cui permessi di accesso vengono determinati da *mode* e dalla mode creation mask del processo

Restituisce: 0 se OK,
-1 in caso di errore

mkdir

- La directory creata avrà come
 - owner ID = l'effective ID del processo
 - group ID = group ID della directory padre
 - vedi altre caratteristiche con `man 2 mkdir`
- La directory sarà vuota ad eccezione di `.` e `..`

rmdir

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
#include <unistd.h>
```

```
int rmdir (const char *pathname);
```

Descrizione: viene decrementato il numero di link al suo i-node; se esso =0 si libera la memoria solo se nessun processo ha quella directory aperta

Restituisce: 0 se OK,
-1 in caso di errore

Funzioni opendir e readdir

```
#include <sys/types.h>
```

```
#include <dirent.h>
```

```
DIR *opendir (const char *pathname);
```

Restituisce il puntatore se OK, NULL in caso di errore

```
struct dirent *readdir (DIR *dp);
```

Restituisce il puntatore se OK, NULL in caso di errore o se alla fine della directory

```
void rewinddir (DIR *dp);
```

```
int closedir (DIR *dp);
```

Struttura dirent

```
struct dirent {  
    ino_t    d_ino;    /* i-node number */  
    .....  
    char     d_name[256]; /* Null-terminated filename */  
};
```

La struttura contiene altri campi non facenti parte di Posix1 e non contenuti in tutti i sistemi.

Funzioni `chdir`, `fchdir`

```
#include <unistd.h>
```

```
int chdir (const char *pathname);
```

```
int fchdir (int fd);
```

Descrizione: cambiano la cwd del processo chiamante a quella specificata come argomento

Restituiscono: 0 se OK
-1 in caso di errore

►► Si noti che:

- cwd è un attributo del processo
- home directory è un attributo di una login name

getcwd

```
#include <unistd.h>
```

```
char *getcwd (char *buf, size_t size);
```

Descrizione: ottiene in *buf* il path assoluto della cwd

Restituisce: *buf* se OK

NULL in caso di errore

Esercizio 3_1

Creare un link simbolico con il comando **ln -s**.

Scrivere un programma che cerchi un link simbolico nella directory e stampi a video il contenuto del file a cui punta il link.

Esercizio 3_2

Scrivere un programma che con un loop, crea una directory e poi va in questa nuova directory.

A quale profondità si riesce ad arrivare?