Laboratorio di sistemi operativi
A.A. 2010/2011
Gruppo 2
Gennaro Oliva
18
Filesystem

Gennaro Oliva
gennaro.oliva@cnr.it
http://www.na.icar.cnr.it/~oliva

Delayed write

- Per motivi di efficienza, il kernel dei sistemi UNIX gestisce le scritture su disco utilizzando un'area buffer, in memoria centrale in cui scrive le operazioni di I/O pendenti
- Questo sistema viene chiamato delayed write
- In un secondo momento il kernel accede all'area buffer scrivendo le informazioni pendenti sul disco per esempio ha bisogno di liberare e riutilizzare il buffer

sync ed fsync

- Esistono però tre system call che consentono di richiedere esplicitamente al kernel una sincronizzazione tra il buffer di memoria per l'I/O e i dati scritti sul disco

  void sync(void);

- richiede di dare inizio alla scrittura senza attendere che sia completata

  int fsync(int fd);

- richiede la scrittura del file corrispondente a fd e attende che la scrittura su disco sia stata completata

fdatasync

- int fdatasync(int fd);
- fdatasync attende che i dati siano scritti ma non si preoccupa degli attributi del file
- Se il disco ha una cache interna le primitive di sincronizzazione attendono che le informazioni siano state scritte nella cache e non sul disco
- In fase di apertura di un file è possibile utilizzare il flag O_SYNC in modo tale che ogni write attenda che la scrittura su disco sia completata

stat, fstat e lstat

- Ad ogni file sono associate una serie di informazioni o metadati quali la dimensione, i permessi, le date di accesso e di modifica
- Queste informazioni possono essere lette dal filesyste e memorizzate in una struttra dati chiamata stat così definita:

```
stastruct stat {
    mode_t      st_mode;    /*tipo di file e permessi */
    uid_t       st_uid;     /* UID del proprietario */
    gid_t       st_gid;     /* GID del propietario */
    ino_t       st_ino;     /* numero dell'inode */
    dev_t       st_dev;     /* numero del device */
    dev_t       st_rdev;    /* tipo di device */
    nlink_t     st_nlink;   /* numero di link */
    off_t       st_size;    /* dimensione totale */
    time_t      st_atime;   /* orario dell'ultimo accesso */
    time_t      st_mtime;   /* orario dell'ultima modifica */
    time_t      st_ctime;   /* orario dell'ultimo cambiamento */
    blksize_t   st_blksize; /* blocksize  */
    blkcnt_t    st_blocks;  /* numero dei blocchi allocati */
};
```

stat, fstat e lstat

- Queste informazioni sui file possono essere lette utilizzando:

```
stat(const char *pathname, struct stat *info);
```

- che fornisce informazioni sul file pathname

  int fstat(int fd, struct stat *info);

- che fornisce tutte le informazioni sul file aperto corrispondente al file descriptor fd

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

- che funziona come fstat ma riconosce i link simbolici
- Tutte restituiscono `0` (successo) o `-1` (errore)

mode_t

- Analizziamo il primo campo della

```
struct stat {
    mode_t      st_mode;    /*tipo di file e permessi */
    uid_t       st_uid;     /* UID del proprietario */
    gid_t       st_gid;     /* GID del propietario */
    ino_t       st_ino;     /* numero dell'inode */
    dev_t       st_dev;     /* numero del device */
    dev_t       st_rdev;    /* tipo di device */
    nlink_t     st_nlink;   /* numero di link */
    off_t       st_size;    /* dimensione totale */
    time_t      st_atime;   /* orario dell'ultimo accesso */
    time_t      st_mtime;   /* orario dell'ultima modifica */
    time_t      st_ctime;   /* orario dell'ultimo cambiamento */
    blksize_t   st_blksize; /* blocksize  */
    blkcnt_t    st_blocks;  /* numero dei blocchi allocati */
};
```

- Questo campo memorizza il tipo di file e i permessi

Tipi di file

- File regolari – contengono dati, senza distinzione tra file binari o di testo
- Directory – Contiene nomi di file e puntatori ad i-node
- Block special device – Un file speciale che fornisce un'interfaccia bufferizzata a blocchi di lunghezza fissa per l'I/O da dispositivi di memorizzazione quali hard disk, cdrom, floppy ...
- Character special device – Un file speciale che fornisce un interfaccia non bufferizzata con blocchi di lunghezza variabile a dispositivi quali schede audio, terminali, porte seriali, ...
- FIFO – Un tipo di file destinato alla comunicazione interprocesso

Tipi di file

- Socket – Utilizzati per comunicazione tra processi anche su rete
- Link simbolici – Puntatori a file

Macro per identificare il tipo di file

- Il tipo di file associato ad un pathname od un file descriptor è memorizzato insieme ai permessi nel campo st_mode della struttura stat
- Le seguenti macro consentono di identificare il tipo di file in esame analizzando il campo st_mode della struttura stat passato come parametro e restituendo `1` se il file in esame corrisponde al tipo associato e `0` altrimenti:
- S_ISREG(m): test per file regolari;
- S_ISDIR(m): test per directory;
- S_ISCHR(m): test per character device;
- S_ISBLK(m): test per block device;
- S_ISFIFO(m): test per fifo;
- S_ISLNK(m): test per link simbolico;
- S_ISSOCK(m): test per socket

Esempio d'uso del camp st_mode

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[]) {
  struct stat permission;
  if ( argc < 2 ) exit(1);
  if (lstat (argv[1], &permission) < 0)
    {
      perror ("lstat");
      return 1;
    }
  printf ("File %s \n", argv[1]);
         if (S_ISREG(buf.st_mode))
            ptr = "regular";
         else if (S_ISDIR(buf.st_mode))
            ptr = "directory";
         else if (S_ISCHR(buf.st_mode))
            ptr = "character special";
         else if (S_ISBLK(buf.st_mode))
            ptr = "block special";
         else if (S_ISFIFO(buf.st_mode))
            ptr = "fifo";
         else if (S_ISLNK(buf.st_mode))
            ptr = "symbolic link";
         else if (S_ISSOCK(buf.st_mode))
            ptr = "socket";
         else
            ptr = "** unknown mode **";
         printf("%s\n", ptr);
  }
   exit(0);
}
```

st_mode e permessi

- Il campo st_mode contiene anche le informazioni sui permessi dei file ed è composto dall'OR delle costanti:
- S_IRUSR: r per il proprietario
- S_IWUSR: w per il proprietario
- S_IXUSR: x per il proprietario
- S_IRGRP: r per il gruppo
- S_IWGRP: w per il gruppo
- S_IXGRP: x per il gruppo
- S_IROTH: r per gli altri
- S_IWOTH: w per gli altri
- S_IXOTH: x per gli altri
- Per sapere se un determinato bit è impostato si può utilizzare l'operatore &

Esempio d'uso del campo st_mode

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[]) {
  struct stat permission;
  if ( argc < 2 ) exit(1);
  if (lstat (argv[1], &permission) < 0)
    {
      perror ("lstat");
      return 1;
    }
  printf ("File %s \n", argv[1]);
  if (permission.st_mode & S_IRUSR)
    printf ("Lettura per proprietario\n");
  if (permission.st_mode & S_IWUSR)
    printf ("Scrittura per proprietario\n");
  if (permission.st_mode & S_IXUSR)
    printf ("Esecuzione per proprietario\n");
  if (permission.st_mode & S_IRGRP)
    printf ("Lettura per gruppo\n");
  if (permission.st_mode & S_IWGRP)
    printf ("Scrittura per gruppo \n");
  if (permission.st_mode & S_IXGRP)
    printf ("Esecuzione per gruppo \n");
  return 0;
}
```

st_uid, st_gid

- Ad ogni file sono associati uno User ID (uid) ed un Group ID (gid) memorizzati in st_uid e st_gid della struttura stat
- struct stat {

```
    mode_t      st_mode;    /*tipo di file e permessi */
    uid_t       st_uid;     /* UID del proprietario */
    gid_t       st_gid;     /* GID del propietario */
    ino_t       st_ino;     /* numero dell'inode */
    dev_t       st_dev;     /* numero del device */
    dev_t       st_rdev;    /* tipo di device */
    nlink_t     st_nlink;   /* numero di link */
    off_t       st_size;    /* dimensione totale */
    time_t      st_atime;   /* orario dell'ultimo accesso */
    time_t      st_mtime;   /* orario dell'ultima modifica */
    time_t      st_ctime;   /* orario dell'ultimo cambiamento */
    blksize_t   st_blksize; /* blocksize  */
    blkcnt_t    st_blocks;  /* numero dei blocchi allocati */
};
```

Real ed effective user e group

- A ogni processo sono associati due utenti
  - Real user: utente che ha lanciato il processo
  - Effective user: utente che determina i diritti del processo
- quando il processo accede al filesystem, il kernel verifica i permessi relativi all'effective user
- Solitamente i due utenti coincidono, ma quando un file eseguibile ha il set-uid-bit impostato, il corrispondente processo avrà:
  - Real user: utente che ha lanciato il processo
  - Effective user: utente proprietario dell'eseguibile
- Analogamente esiste una distinzione tra effective group e real group ed un set-gid-bit

Accesso di un processo ad un file

- L'accesso ad un file da parte di un processo dipende dall'effective user e gid del processo ed è regolato dalla seguente sequenza di controllo:
- Se l'effective user del processo e' 0 (root): OK
- Se l'effective user del processo e' uguale all'owner del file
  - Controlla i permessi per l'utente ed, in caso, nega l'accesso
- Se l'effective group del processo e' uguale al group del file
  - Controlla i permessi del gruppo ed, in caso, nega l'accesso
- Altrimenti
  - Controlla i permessi per gli “altri”
- Se l'effective uid è anche il proprietario del file ma non ha i permessi per leggerlo l'accesso viene negato anche se il suo gruppo o gli altri hanno il permesso

access

- In alcuni casi, però, un programma potrebbe avere la necessità di verificare le protezioni in base al real user/groud id
- Nel caso del comando passwd, è necessario sapere se il real user è diverso da root per impedirgli eventualmente di cambiare la password di altri utenti
- Questa verifica può essere effettuata esplicitamente all'interno del codice utilizzando:

  int access(const char *pathname, int mode);
- Il parametro mode può assumere i valori: R_OK, W_OK, X_OK per verificare rispettivamente, l'accesso in lettura, scrittura od esecuzione oppure F_OK se il test riguarda l'esistenza del file
- La system call restituisce `0` se i permessi del file indicato dal parametro pathname consentono l'accesso al processo corrente in base a real user id e real group id
- Il processo continua ad avere i permessi dell'effective user id ma il codice potrà specificare diverse modalità di funzionamento a seconda del valore restituito da access

Utilizzo “improprio” di access

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
  if (argc != 2) exit(1);
  if (access(argv[1], R_OK) < 0)
    printf("no read access %s\n", argv[1]);
  else
    printf("read access %s\n", argv[1]);
  if (open(argv[1], O_RDONLY) < 0)
    printf("open error for %s\n", argv[1]);
  else
    printf("open for reading OK\n");
        exit(0);
}
```

Uso “improprio” di access

- ```
  $ ls -l a.out /etc/shadow
  -rwxr-xr-x 1 oliva oliva  5088 May 12 09:38 a.out
  -rw-r----- 1 root  shadow 1319 Feb 17 08:46 /etc/shadow
  $ ./a.out a.out
  read access OK
  open for reading OK
  $ ./a.out /etc/shadow
  access error for /etc/shadow
  open error for /etc/shadow
  $ su
  password:
  # chown root.root a.out ; chmod u+s a.out ; exit
  $ ls -l a.out
  -rwsr-xr-x 1 root root 5088 May 12 09:40 a.out
  $ ./a.out /etc/shadow
  access error for /etc/shadow
  open for reading OK
  ```
  ```

chown, fchown e lchown

- È possibile modificare il proprietario ed il gruppo di un file utilizzando:

```
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);
```

- che modificano il proprietario ed il gruppo di appartenenza del file indicato dal pathname (chown e lchown) o dal file descriptor (fchown)
- Nel caso in cui il pathname identifica un link simbolico, la lchown modifica il proprietario/gruppo del link piuttosto che del file puntato dal link
- Se il parametro “owner” o “group” e' uguale a -1, il campo corrispondente non viene modificato; questo consente di modificare uno dei due elementi senza conoscere l’altro

chown, fchown e lchown

- In molti sistemi, solo un processo di root può modificare il proprietario del file
- Il processo di un utente regolare può utilizzare le funzioni chown per modificare il gruppo di un file se:
- (a) l'effective user id del processo è pari all'owner id del file
- (b) la system call lascia il proprietario del file inalterato
- (c) il parametro group e' uguale all'effective GID del processo oppure ad uno dei gruppi supplementari

struct stat

- Le informazioni relative alla dimensione del file sono contenute nei campi st_size, st_blksize, st_blocks
- struct stat {

```
    mode_t      st_mode;    /*tipo di file e permessi */
    uid_t       st_uid;     /* UID del proprietario */
    gid_t       st_gid;     /* GID del propietario */
    ino_t       st_ino;     /* numero dell'inode */
    dev_t       st_dev;     /* numero del device */
    dev_t       st_rdev;    /* tipo di device */
    nlink_t     st_nlink;   /* numero di link */
    off_t       st_size;    /* dimensione totale */
    time_t      st_atime;   /* orario dell'ultimo accesso */
    time_t      st_mtime;   /* orario dell'ultima modifica */
    time_t      st_ctime;   /* orario dell'ultimo cambiamento */
    blksize_t   st_blksize; /* blocksize  */
    blkcnt_t    st_blocks;  /* numero dei blocchi allocati */
};
```

st_size

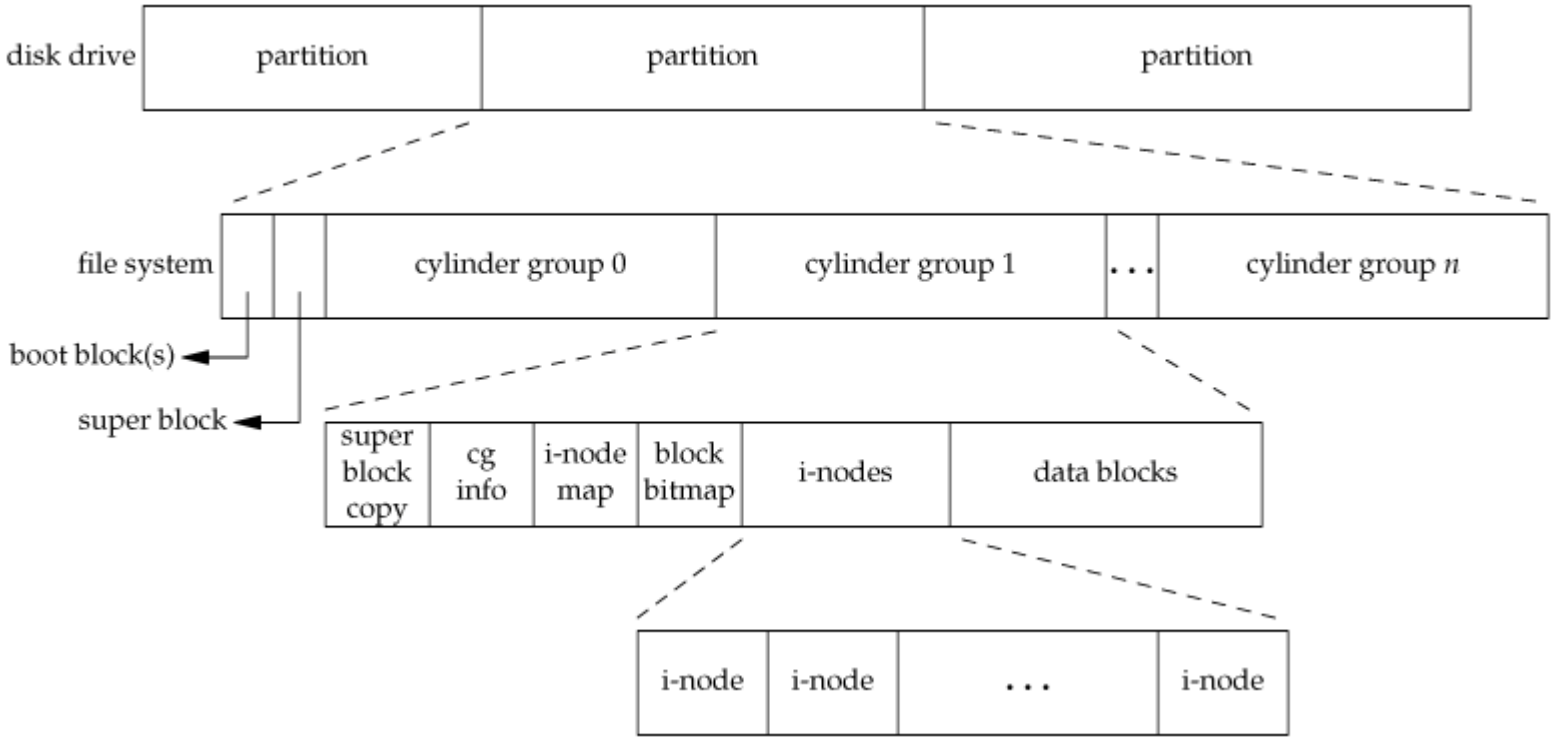
- Il campo st_size della struttura struct contiene la dimensione in byte del file
- Quest'informazione ha senso solo per file regolari, directory e link per cui assume i seguenti significati
- file regolari: numero di byte contenuti nel file;
- directory è solitamente un multiplo di 16 o 512;
- link: il numero di byte del pathname del file puntato dal link
- Il campo st_blksize contiene la dimensione “ottimale” da utilizzare quando si effettuano operazioni di I/O (per ottimizzare le performance di lettura e scrittura)
- Il campo st_blocks indica il numero di blocchi allocati per il file

truncate ed ftruncate

- ```
  #include <unistd.h>
  #include <sys/types.h>
  int truncate(const char *path, off_t length);
  int ftruncate(int fd, off_t length);
  ```
- Queste funzioni “troncano” rispettivamente il file specificato dal pathname passato come primo argomento o il file aperto associato al file descriptor a fd dopo “length” bytes
- Se la dimensione del file è maggiore del parametro length, i dati dal byte length+1 non saranno più accessibili
- Se la dimensione e' minore di length, il comportamento dipende dall'implementazione
  - Nei sistemi Linux, la dimensione del file viene aumentata e la parte in eccesso contiene “zeri”
- Le system call restituiscono: `0` in caso di successo e `-1` in caso di errore
  ```

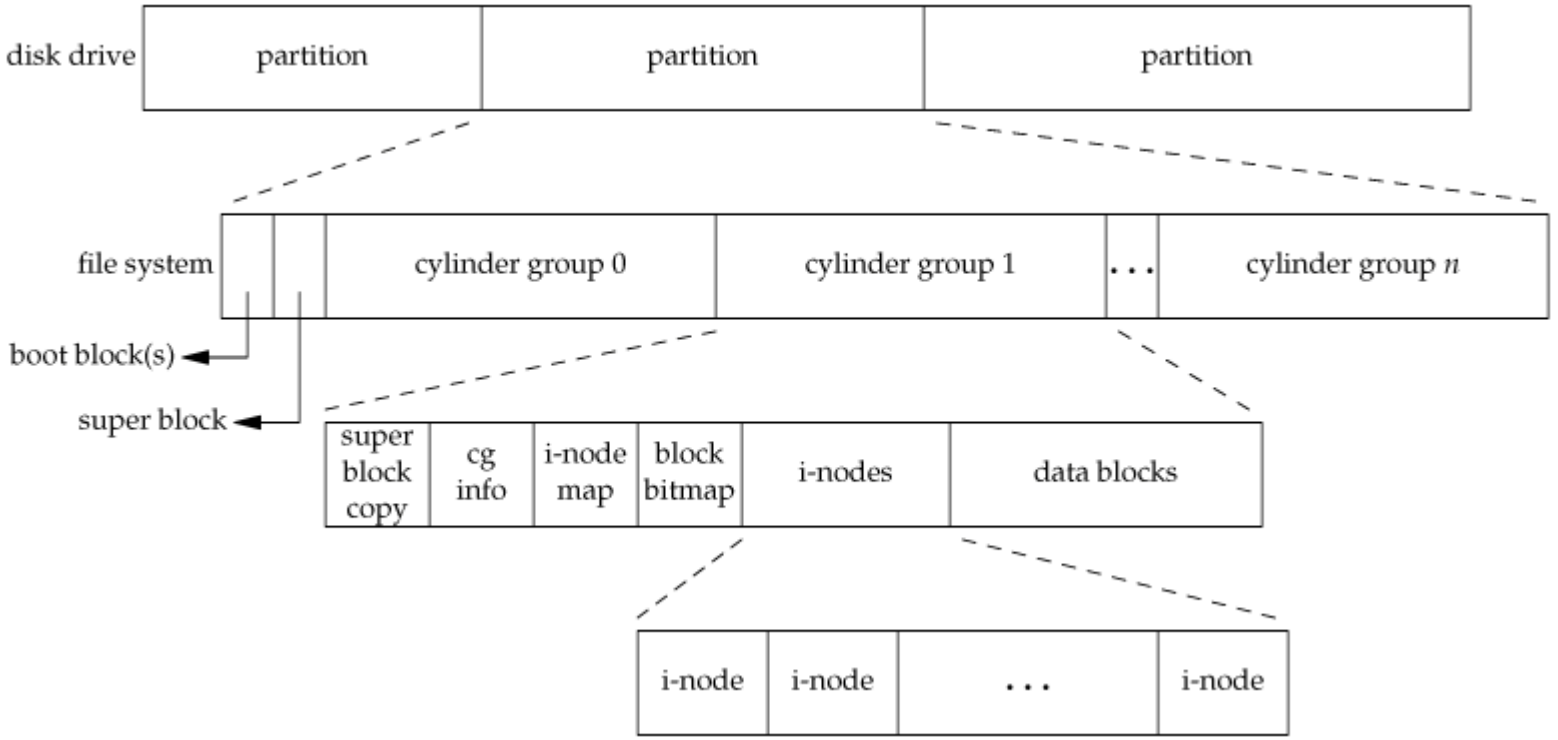
Partizionamento di un disco

- Ogni disco è diviso in una o più partizioni
- All'interno di ciascuna partizione c'e' un boot block dove viene memorizzato il codice per l'avvio del systema operativo, un super block in cui vengono memorizzate le informazioni sul filesystem ed n cylinder group



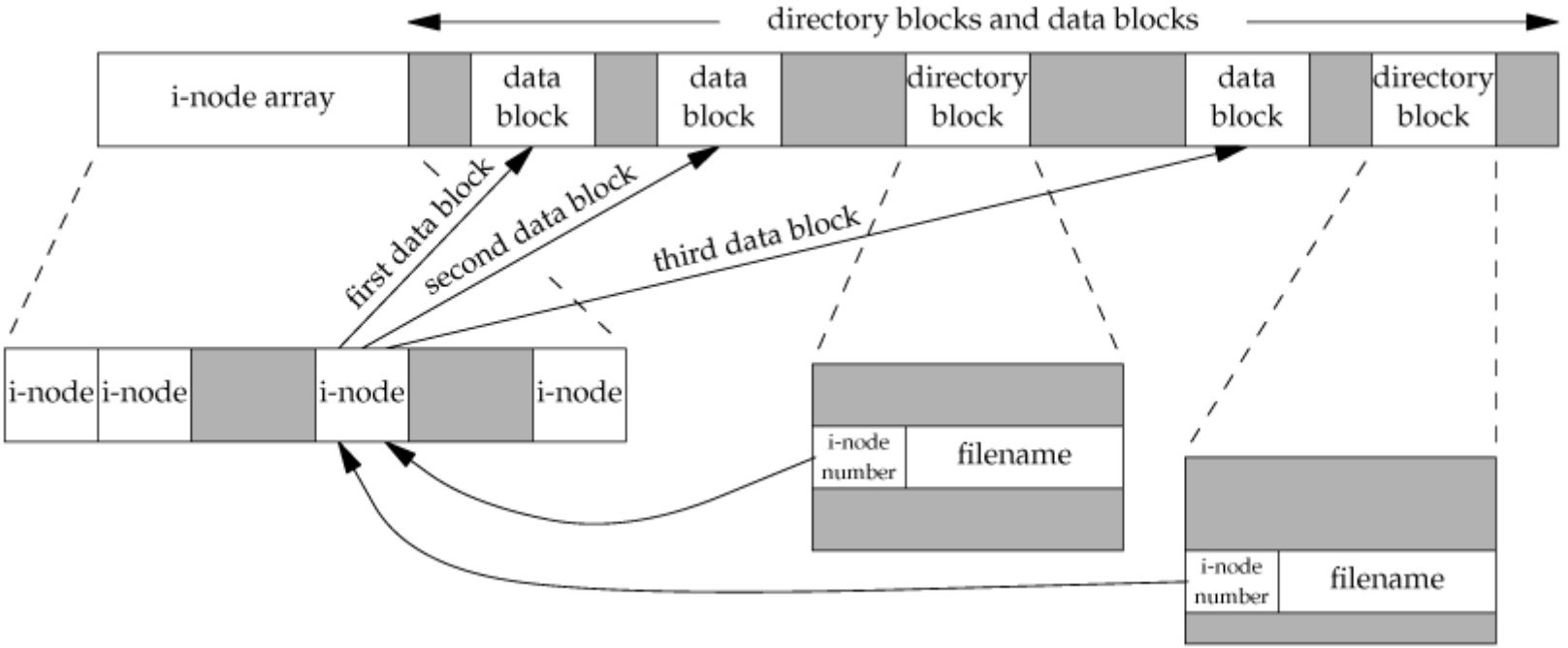
Partizionamento di un disco

- All'inizio di ciascun cylinder group possiamo trovare una copia del super block del filesystem, le informazioni sul cilinder group e le mappe sugli i-node ed i data block allocati
- Successivamente troveremo gli i-node e i data block



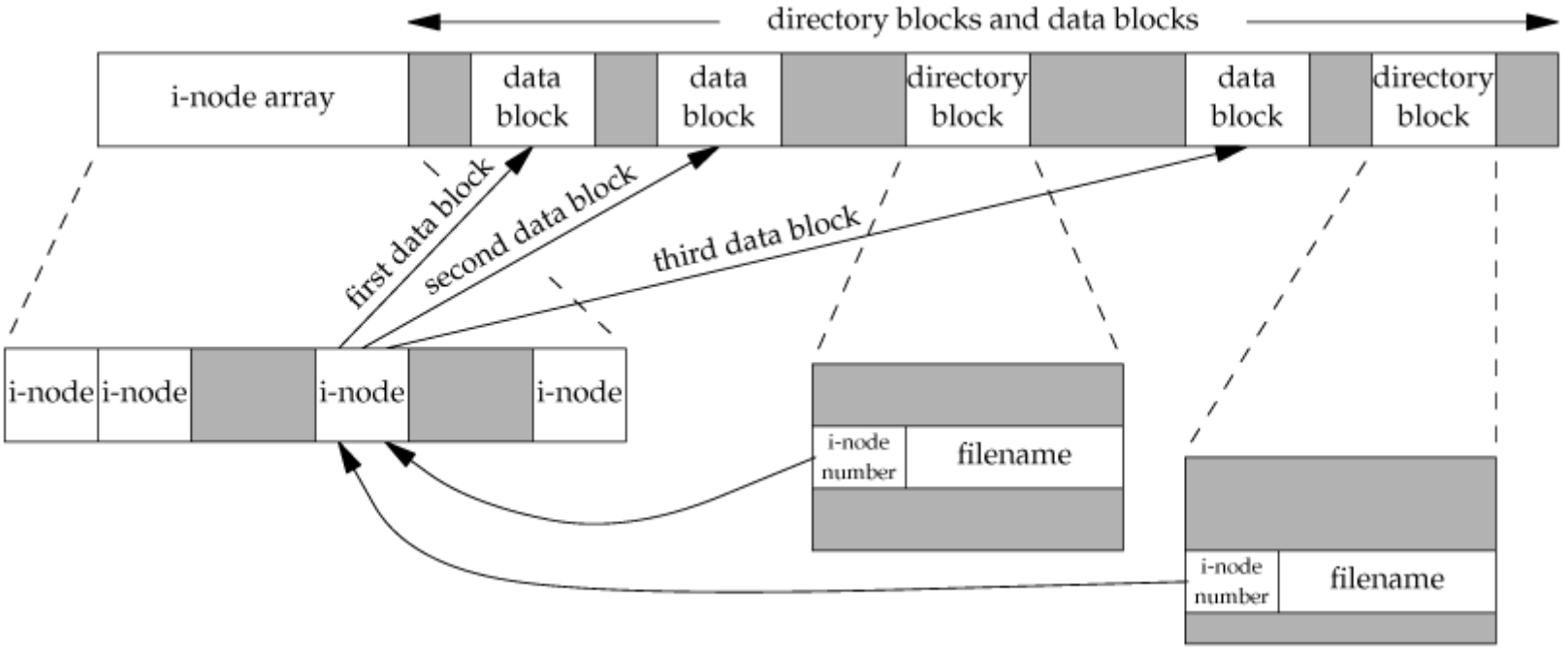
i-node

- Ad ogni file è associato un singolo e unico i-node che memorizza la maggior parte delle informazioni restituite da stat
- Il contenuto del file viene memorizzato in zero o più data block
- I puntatori ai data block sono memorizzati all'interno dell'i-node
- All'aumento delle dimensioni del file può corrispondere un aumento dei data block allocati per il file



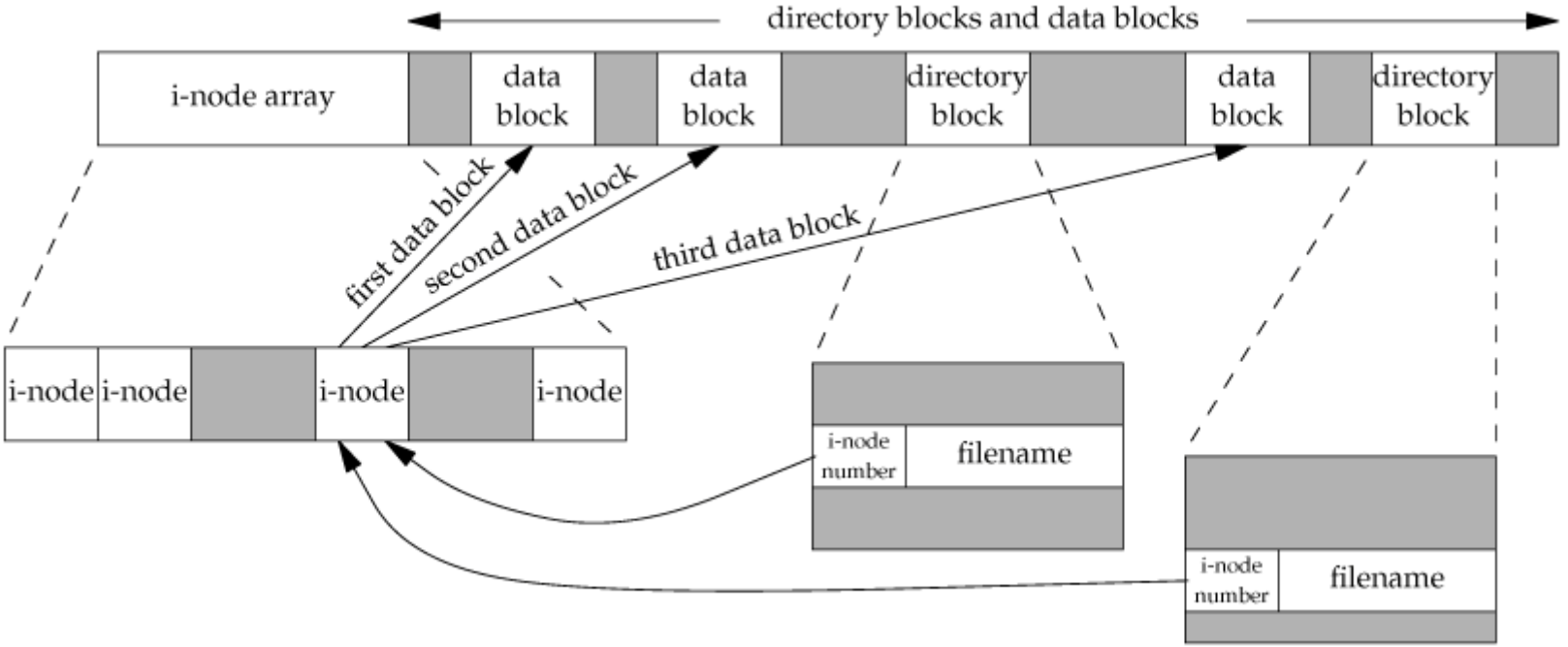
Directory block

- Anche alle directory come a qualsiasi altro tipo di file presente sul filesystem è associato un singolo e unico i-node
- Il data block di una directory viene detto directory block e contiene per ogni file contenuto nella directory le seguenti informazioni:
  - il nome del file
  - il numero di i-node ad esso associato
- Si noti che la modifica del nome di un file viene effettuata sull'informazione memorizzata nel directory block



Hard link

- Un hard link è un elemento all'interno di una directory il cui numero di i-node corrisponde ad quello di un altro file
- In pratica è possibile creare "un file" all'interno di una directory creando un puntatore ad un file già esistente e non c'e' modo di distinguere tra l'hard link e il file originale



Soft link o symbolic link

- Un soft link (o symbolic link) è una versione di link più duttile
- Un soft link corrisponde ad un file che contiene, all'interno dell'unico data block ad esso associato, il nome di un altro file
- Un soft link può essere utilizzato per creare puntatori a file o directory lasciando sempre ben chiara la distinzione tra il file originale e il link

Limiti per gli Hard Link

- La maggior parte di implementazioni UNIX moderne non consente di creare hard link a directory
- Questo limite viene imposto per evitare ricorsioni infinite nel tentativo di attraversare un albero che parte da un'hard link alla propria directory padre
- Non è possibile effettuare un hard link con un file memorizzato su una partizione diversa da quella della directory in cui lo si vuole creare

struct stat

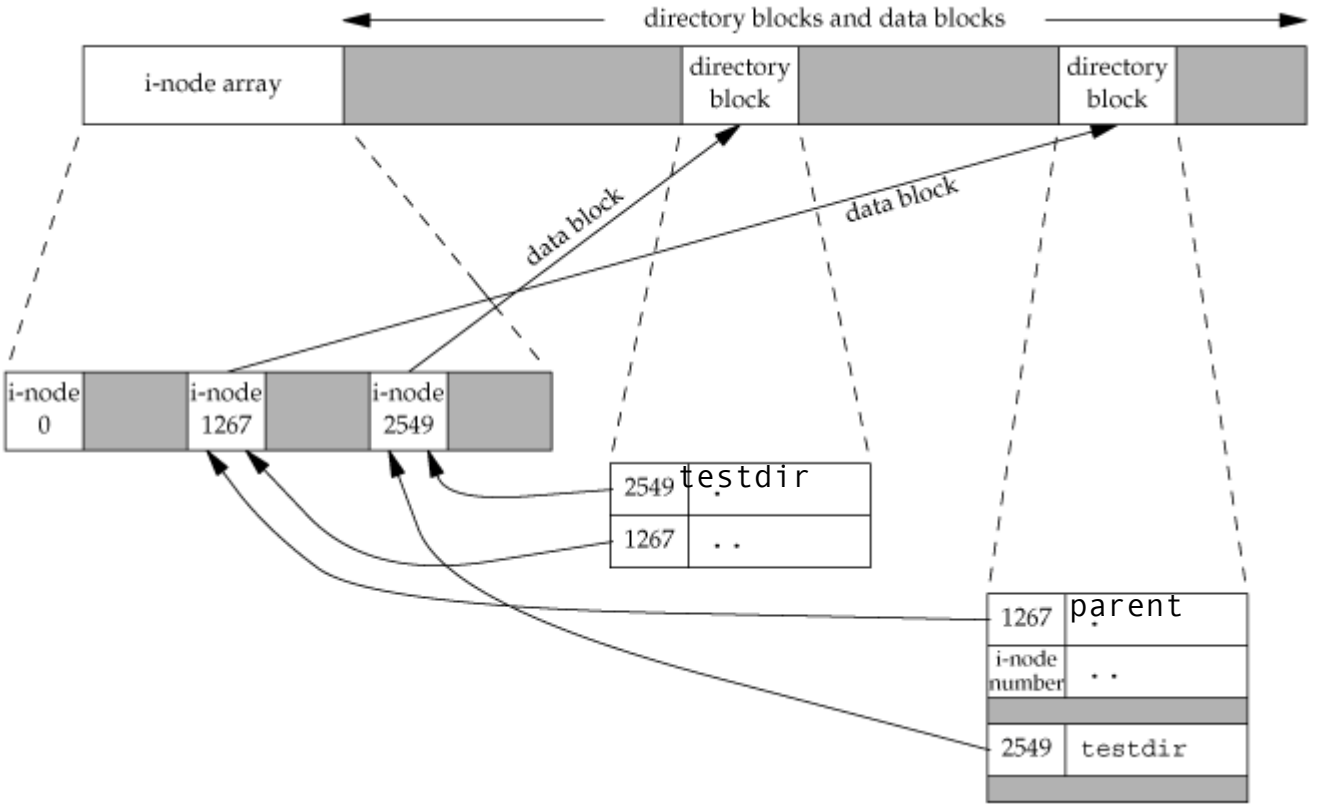
- Il numero di hard link al file è accessibile dal campo
- struct stat {

```
    mode_t      st_mode;    /*tipo di file e permessi */
    uid_t       st_uid;     /* UID del proprietario */
    gid_t       st_gid;     /* GID del propietario */
    ino_t       st_ino;     /* numero dell'inode */
    dev_t       st_dev;     /* numero del device */
    dev_t       st_rdev;    /* tipo di device */
    nlink_t     st_nlink;   /* numero di link */
    off_t       st_size;    /* dimensione totale */
    time_t      st_atime;   /* orario dell'ultimo accesso */
    time_t      st_mtime;   /* orario dell'ultima modifica */
    time_t      st_ctime;   /* orario dell'ultimo cambiamento*/
    blksize_t   st_blksize; /* blocksize  */
    blkcnt_t    st_blocks;  /* numero dei blocchi allocati */
};
```

- Un file appena creato ha sempre un solo hard link

Link count per directory

- Le directory hanno un numero di link almeno 2:
  - un puntatore è memorizzato nella voce corrispondente contenuta dalla directory “padre”
  - un puntatore è contenuto nella stessa directory verso “.”



link, unlink

- Per creare un hard link è possibile utilizzare:

  int link(const char *oldpath, const char *newpath);
- che crea un (hard) link tra oldpath (file esistente) e newpath e restituisce `0` se l'operazione ha successo e -1 in caso di errore
- Per rimuovere un hard link è possibile utilizzare:

  int unlink(const char *pathname);
- che rimuove dal directory block il riferimento al file decrementa il numero di hard link nell'i-node del file
- Nel caso tale numero diventi zero elimina il file solo se non vi sono altri processi che lo tengono aperto
- Per utilizzare unlink sono necessari permessi di scrittura e esecuzione sulla directory o, se lo sticky bit è settato, essere proprietari della directory o del file

sticky bit

- Se una directory ha lo sticky bit impostato i file e subdirectory in essa contenuti possono essere cancellati o rinominati solo dal proprietario del file o dal proprietario della directory che lo contiene o dall'utente root
- Lo sticky bit viene utilizzato per le directory dove vengono creati e cancellati i file temporanei di più utenti come /tmp e /var/tmp per evitare che utenti regolari cancellino o spostino i file temporanei di altri utenti, pur consentendo a chiunque di creare nuovi file e directory

```
$ ls -lad /tmp
drwxrwxrwt 40 root root 12288 May 19 10:45 /tmp/
```

rename

- È possibile modificare il path name associato ad un file utilizzando:

```
int rename(const char *oldpath, const char *newpath);
```

- Rinomina il file indicato da oldpath come newpath secondo le seguenti regole:
  - Se oldpath non è una directory, newpath non può essere una directory
  - Se newpath esiste, viene sovrascritto
  - Se oldpath è una directory
    - newpath deve essere una directory vuota
    - newpath non può contenere oldpath come prefisso (non è possibile rinominare /usr come /usr/bin)

Soft link (o link simbolici)

- I soft link (o link simbolici) superano i limiti degli hard link in quanto consentono di:
  - creare collegamenti tra entità su filesystem diversi
  - Creare link a directory
- La directory entry per un link contiene:
  - Nel caso di hard link: il numero di inode del file puntato
  - Nel caso I soft link: il nome del file puntato
- Se si opera su link simbolici e si passano come argomenti alle funzioni o alle system call è necessario controllare sempre se la funzione “segue” il link (rename di un symbolic link non segue il link)
- Per la gestione dei soft link è possibile utilizzare:

  int symlink(const char *oldpath, const char *newpath);
- Per crea un link simbolico tra oldpath e newpath (non è necessario che oldpath esista)

  int readlink(const char *path, char *buf, size_t bufsiz);
- legge il “nome del file” a cui il link punta

Date nella struct stat

- La struttura stat fornisce tre diverse date per ciascun file
- struct stat {

```
    mode_t      st_mode;    /*tipo di file e permessi */
    uid_t       st_uid;     /* UID del proprietario */
    gid_t       st_gid;     /* GID del propietario */
    ino_t       st_ino;     /* numero dell'inode */
    dev_t       st_dev;     /* numero del device */
    dev_t       st_rdev;    /* tipo di device */
    nlink_t     st_nlink;   /* numero di link */
    off_t       st_size;    /* dimensione totale */
    time_t      st_atime;   /* orario dell'ultimo accesso */
    time_t      st_mtime;   /* orario dell'ultima modifica */
    time_t      st_ctime;   /* orario dell'ultimo cambiamento */
    blksize_t   st_blksize; /* blocksize  */
    blkcnt_t    st_blocks;  /* numero dei blocchi allocati */
};
```

Campo	Descrizione	Esempio
st_atime	Ultima data di accesso ai dati	read
st_mtime	Ultima data di modifica dei dati	write
st_ctime	Ultima data di modifica dell'i-node	chmod

Date nella struct stat

- Una modifica delle informazioni contenute nell'i-node (es. permessi del file) comporta una modifica di st_ctime, ma non di st_mtime
- Il sistema non conserva la data di ultimo accesso all'i-node quindi le pertanto le syustem call stat fstat e lstat non modificano alcuna data

Campo	Descrizione	Esempio
st_atime	Ultima data di accesso ai dati	read (non stat)
st_mtime	Ultima data di modifica dei dati	write
st_ctime	Ultima data di modifica dell'i-node	chmod

umask

- La umask (user file mode creation mask) di un processo fornisce uno strumento per la limitazione dei permessi di tutti i file creati dal processo in fase di creazione (e delle directory)
- La umask ci consente di dire quali bit dei permessi vogliamo lasciare sempre a 0 ovvero quali permessi vogliamo sempre negare
- I permessi di accesso al file vengono modificati come segue

```
PERMESSI RICHIESTI AND
NOT MASK            =
--------------------
PERMESSI ATTRIBUITI
```

```
PERMESSI 110110110
MASK     000010010
NOT MASK 111101101
```

```
rwxrwxrwx
110110110 AND
111101101 =
---------
110100100
```

umask

- La creation mask consente di evitare la creazione di file con permessi che potrebbero compromettere la sicurezza dei dati contenuti
- La creation mask di default è visualizzabile e modificabile utilizzando il comando umask
- In molti sistemi è impostata a `022` che corrisponde a `000010010` ovvero S_IWGRP|S_IWOTH
- In questo modo i file creati avranno sempre i permessi di scrittura per gruppo e altri disabilitati

umask

- Per modificare la creation mask per il processo in esecuzione si può utilizzare la system call:

  mode_t umask(mode_t mask);

- Come argomento della system call si passano i permessi da inserire

  S_I{R,W,X}{USR,GRP,OTH}

- concatenati con l'operatore or “|”.
- Esempio d'uso:

  umask(S_IWOTH|S_IWGRP);

Gestione delle directory: mkdir

- Per creare una directory vuota è possibile utilizzare:
- int mkdir(const char *path, mode_t perm);
- Il parametro path è il nome della directory e perm sono i suoi permessi (modificati in base alla user creation mask)
- Per utilizzare la directory è necessario impostare i permessi di esecuzione
- Una directory vuota contiene sempre due sottodirectory: “.” e “..”
- Restituisce `0` in caso di successo -1 altrimenti

Gestione delle directory rmdir

- Per cancellare una directory vuota è possibile utilizzare:

  int rmdir(const char *pathname);

- La directory viene effettivamente cancellata se non è aperta da nessun processo in esecuzione
- Restituisce `0` in caso di successo -1 altrimenti

opendir

- Per leggere il contenuto di una directory è possibile utilizzare:
- DIR *opendir(const char *name);
- In caso di successo, restituisce un puntatore alla directory oppure NULL altrimenti
- Il puntatore ottenuto è l'equivalente del puntatore a FILE utilizzato nella libreria standard
- NON è necessario allocare lo spazio per il tipo DIR e passare alla funzione l'indirizzo di memoria

struct dirent

- La struttura dati che descrive l'elemento contenuto in una directory è una struct chiamata dirent
- Sui sistemi Linux la struct ha i seguenti campi:
- ```
  struct dirent {
    ino_t  d_ino; /* inode number */
    off_t  d_off;
    unsigned short d_reclen;
    unsigned char  d_type;
    char   d_name[256]; /* filename */
  };
  ```
- Lo standard POSIX richiede soltanto d_name
  ```

readdir

- L’accesso sequenziale agli elementi di una directory viene effettuato eseguendo la system call

  struct dirent *readdir(DIR *dp);

- La funzione viene normalmente invocata ripetutamente passandole sempre lo stesso puntatore ed ogni invocazione restituisce un diverso elemento della directory
- L'indirizzo restituito è utilizzabile in lettura e lo spazio per memorizzare la struct NON deve essere allocato
- Quando sono terminati gli elementi della directory readdir restituisce NULL
- Anche in caso d'errore viene restituito il valore NULL, ma in tale evenienza viene settata la variabile errno

errno come controllo positivo

- Il valore di errno è significativo solo quando il valore restituito da una system call o da una funzione di libreria indica la presenza di un errore
- Il valore iniziale della variabile è `0`, ma una volta che è stato cambiato nessuna system call o funzione di libreria riporta il suo valore a `0`
- Per verificare che readdir ha fallito dobbiamo settare errno a zero e quindi verificare quando ci viene restituito NULL che il valore della variabile non sia stato alterato

Esempio d'uso

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <dirent.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
int main(int argc, char *argv[]){
  DIR *targetdir;
  struct dirent *entry;
  if (argc != 2) exit(1);
  if ( ( targetdir=opendir(argv[1]) ) == NULL ) {
    perror("opendir");
    exit(1);
  }
  errno=0;
  while ( ( entry = readdir(targetdir) ) )
    printf("%s\n",entry->d_name);
  if ( errno )
    perror ("readdir");
  closedir(targetdir);
  exit(0);
}
```

rewinddir e closedir

- Per riportare l'indice di lettura al primo elemento si può utilizzare

  void rewinddir(DIR *dp);

- mentre per chiudere una directory si utilizza

  int closedir(DIR *dp);

- Per l'accesso non sequenziale alle directory si possono utilizzare telldir e seekdir