

# Gestione di File e Directory

# Duplicazione di file descriptor

- Un file descriptor puo' essere duplicato utilizzando:

```
int dup (int filedes);
```

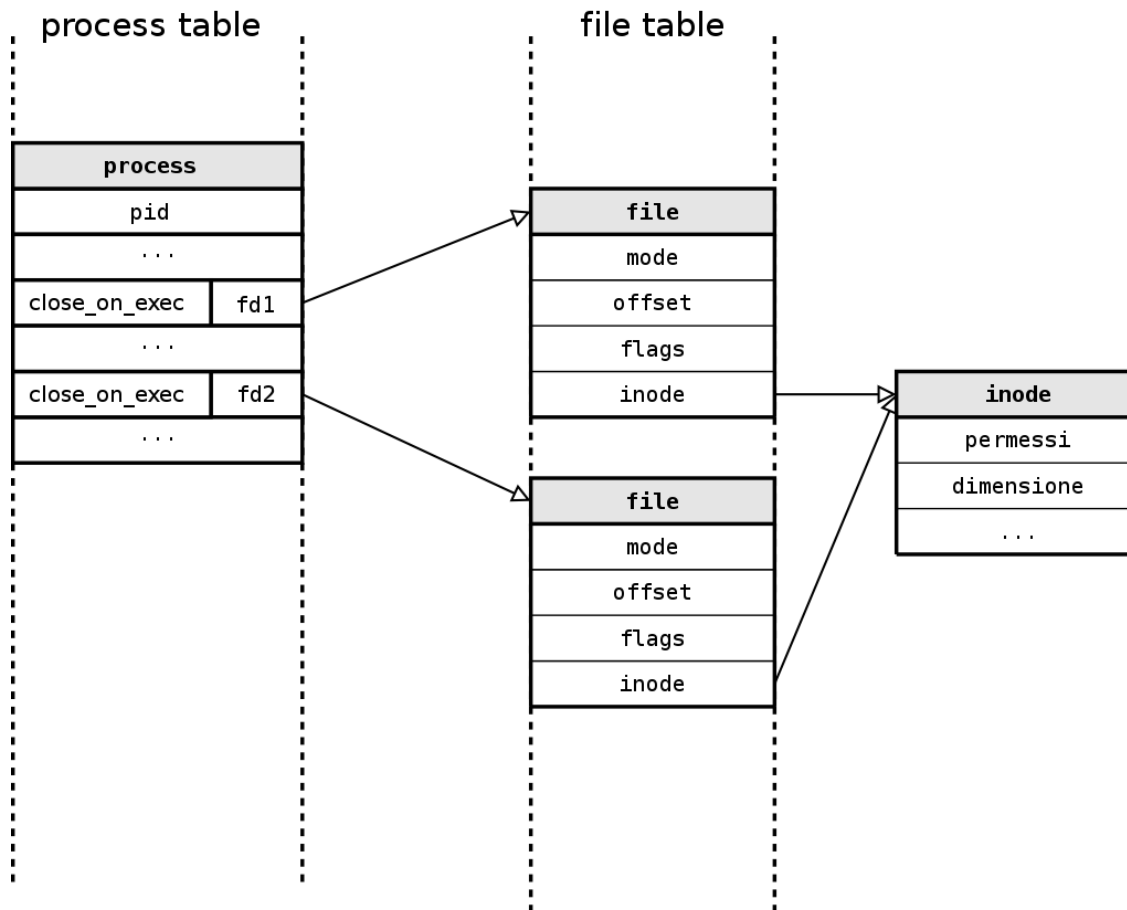
```
int dup2(int filedes, int filedes2);
```

- dup restituisce un file descriptor che punta allo stesso file indirizzato da *filedes*.
  - Il valore ritornato da dup e' il *minimo file descriptor* non utilizzato
- dup2 prende in input *filedes2*, il file descriptor da usare nella duplicazione.
  - Se *filedes2* e' aperto, dup2 chiude il file prima di duplicare il descrittore *filedes*
  - dup2 e' una operazione atomica

# Differenza tra open e dup

```
fd1 = open("file", O_RDONLY);
```

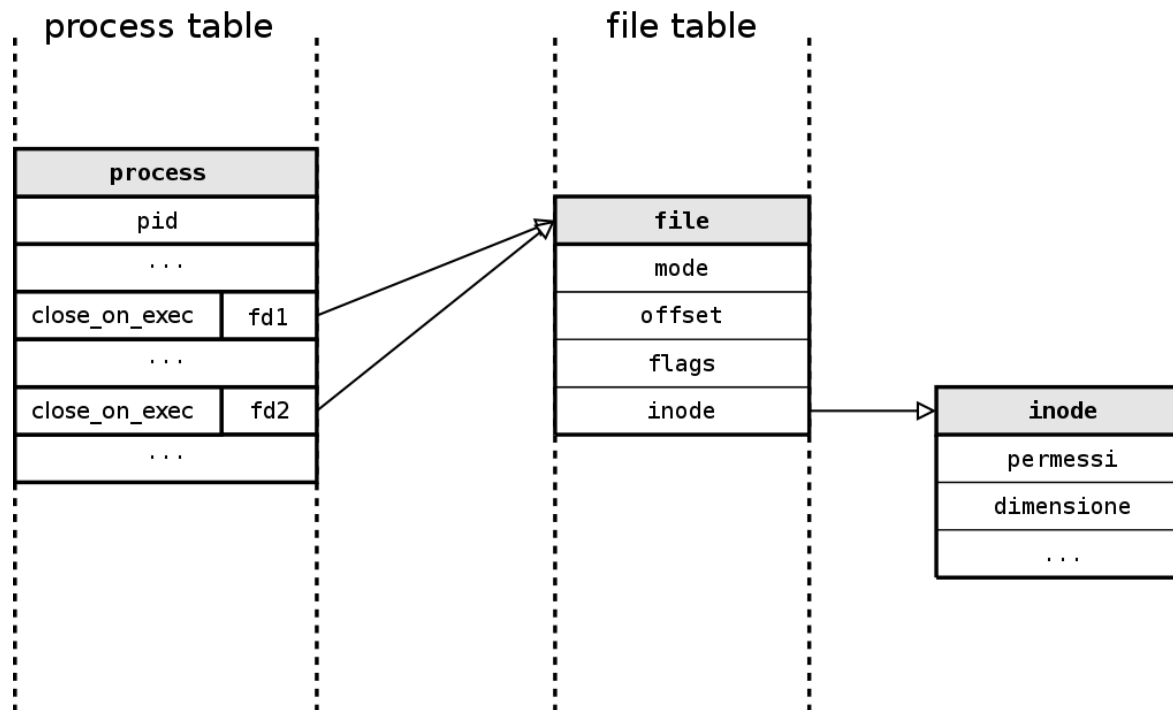
```
fd2 = open("file", O_WRONLY);
```



# Differenza tra open e dup

```
fd1 = open("file", O_RDONLY);
```

```
fd2 = dup(fd1);
```



# Ottenere informazioni su un file

```
int  stat(const char *file_name, struct stat *buf);
int  lstat(const char *file_name, struct stat *buf);
int  fstat(int filedes,          struct stat *buf);
      Valori di ritorno: 0 se OK, -1 su errore.
```

- Queste system call prendono in input un puntatore ad una struttura stat che conterrà le informazioni sul file
- **stat** ed **lstat** prendono in input il nome del file
  - **lstat** dà informazioni sui link simbolici
- **fstat** prende in input il file descriptor del file
  - Il file deve essere aperto.

# La struttura stat

```
struct stat {
    mode_t      st_mode;      /* file type & mode (permissions) */
    uid_t       st_uid;      /* user ID of owner */
    gid_t       st_gid;      /* group ID of owner */
    ino_t       st_ino;      /* inode number */
    dev_t       st_dev;      /* device number (file system) */
    dev_t       st_rdev;     /* device type (if inode device) */
    nlink_t     st_nlink;    /* number of links */
    off_t       st_size;     /* total size, in bytes */
    time_t      st_atime;    /* time of last access */
    time_t      st_mtime;    /* time of last modification */
    time_t      st_ctime;    /* time of last change */
    blksize_t   st_blksize;  /* blocksize for filesystem I/O */
    blkcnt_t    st_blocks;   /* number of blocks allocated */
};
```

# La struttura stat

```
struct stat {  
    mode_t      st_mode;      /* file type & mode (permissions) */  
    uid_t       st_uid;       /* user ID of owner */  
    gid_t       st_gid;       /* group ID of owner */  
    ino_t       st_ino;        /* inode number */  
    dev_t       st_dev;        /* device number (file system) */  
    dev_t       st_rdev;       /* device type (if inode device) */  
    nlink_t     st_nlink;      /* number of links */  
    off_t       st_size;       /* total size, in bytes */  
    time_t      st_atime;      /* time of last access */  
    time_t      st_mtime;      /* time of last modification */  
    time_t      st_ctime;      /* time of last change */  
    blksize_t   st_blksize;    /* blocksize for filesystem I/O */  
    blkcnt_t    st_blocks;     /* number of blocks allocated */  
};
```

# Tipi di file

- Regular file:
  - Contiene “dati” di qualche tipo
  - Attenzione: anche gli *eseguibili* sono “regular file”
- Directory file:
  - Contiene nomi e puntatori a inode
  - E' necessario utilizzare system call specifiche per manipolarlo
- Block Special file:
  - Rappresentano particolari device (per es., dischi)

# Tipi di file

- Character Special file:
  - Rappresentano particolari device (per es., scheda audio)
- FIFO:
  - (o pipe) Utilizzati per comunicazione tra processi
- Socket:
  - Utilizzati per comunicazione tra processi
- Symbolic Link:
  - Link simbolico (o soft)

# Tipi di file

- Il tipo di file associato ad un pathname od un file descriptor e' codificato nel campo **st\_mode** della struttura **stat**.
- Per interpretare **st\_mode**, si usano le seguenti macro:
  - S\_ISREG(m): Is regular file ?
  - S\_ISDIR(m): Is directory?
  - S\_ISCHR(m): Is character device?
  - S\_ISBLK(m): Is block device?
  - S\_ISFIFO(m): Is Fifo?
  - S\_ISLNK(m): Is symbolic link?
  - S\_ISSOCK(m): Is socket?
- Le macro prendono come argomento il campo **st\_mode**

# Esempio

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

    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if (lstat(argv[i], &buf) < 0) {
            perror("lstat error");
            continue;
        }
        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";
        ...
        printf("%s\n", ptr);
    }
    return 0;
}
```

# La struttura stat

```
struct stat {
    mode_t      st_mode;      /* file type & mode (permissions) */
    uid_t       st_uid;      /* user ID of owner */
    gid_t       st_gid;      /* group ID of owner */
    ino_t       st_ino;      /* inode number */
    dev_t       st_dev;      /* device number (file system) */
    dev_t       st_rdev;     /* device type (if inode device) */
    nlink_t     st_nlink;    /* number of links */
    off_t       st_size;     /* total size, in bytes */
    time_t      st_atime;    /* time of last access */
    time_t      st_mtime;    /* time of last modification */
    time_t      st_ctime;    /* time of last change */
    blksize_t   st_blksize;  /* blocksize for filesystem I/O */
    blkcnt_t    st_blocks;   /* number of blocks allocated */
};
```

# User e Group ID

- Ad ogni *file* sono associati uno User ID (uid) ed un Group ID (gid)
  - memorizzati in `st_uid` e `st_gid` della struttura `stat`.
- Si ricorda che ogni *processo* possiede i seguenti ID:
  - Real user e Real group:
    - Utente (e gruppo) che ha lanciato il processo
  - Effective user ed Effective group
    - Utente (e gruppo) che determina i diritti di accesso del processo

# Accesso ai file

- Il campo **st\_mode** codifica i permessi di accesso ai file
- Per accedere ad un file e' necessario:
  - avere diritto di **esecuzione** su TUTTE le directory nel path
    - e.g., /home/utente/LSO/esempio.txt
    - il permesso di **lettura** sulla directory consente di leggere i nomi dei file ma non di aprire un file
  - avere i permessi di accesso specifici per il file
- Per creare un file in una directory
  - permessi di scrittura sulla directory
- Per cancellare un file in una directory
  - permessi di scrittura sulla directory (non sul file!)

# Accesso ai file

- L'accesso ai file e' regolato dalla seguente sequenza:
  - Se l'effective user del processo e' 0 (superuser): 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”.

# La funzione access

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

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

Restituisce 0 se OK, -1 su errore

- controlla i permessi di accesso ad un file in base ad UID e GID “reali”
  - mentre normalmente valgono UID e GID effettivi
- il parametro mode puo' assumere i valori
  - R\_OK, W\_OK, X\_OK: Lettura, scrittura o esecuzione
  - F\_OK: Esistenza

# Esempio

```
int main(int argc, char *argv[]){
    if (argc != 2){
        printf("usage: a.out <pathname>\n");
        return 1;
    }
    if (access(argv[1], R_OK) < 0)
        printf("access error for %s\n", argv[1]);
    else
        printf("read access OK\n");
    if (open(argv[1], O_RDONLY) < 0)
        printf("open error for %s\n", argv[1]);
    else
        printf("open for reading OK\n");
    return 0;
}
```

# Esempio:

```
lso:~>ls -l /etc/shadow
```

```
-r-----      1 root    root    1054 Mar  6 21:09 /etc/shadow
```

```
lso:~>ls -l a.out
```

```
-rwxrwxr-x     1 lso      lso     12049 Apr 12 14:38 a.out
```

```
lso:~>./a.out a.out
```

```
read access OK
```

```
open for reading OK
```

```
lso:~>./a.out /etc/shadow
```

```
access error for /etc/shadow
```

```
open error for /etc/shadow
```

```
lso:~> su; chmod u+s a.out; chown root.root a.out
```

```
lso:~> ls -l a.out
```

```
-rwsrwx--x     1 root      root    12049 Apr 12 14:38 a.out
```

```
lso:~>./a.out /etc/shadow
```

```
access error for /etc/shadow
```

```
open for reading OK
```

# chmod e fchmod

```
#include <sys/types.h>
#include <sys/stat.h>
int  chmod(const char *path, mode_t mode);
int  fchmod(int fildes,      mode_t mode);
```

- Consentono di modificare i permessi di accesso ai file
  - chmod prende in input un pathname
  - fchmod prende in input un file descriptor (file deve essere aperto).
- Il parametro “mode” puo' essere una combinazione delle seguenti costanti:
  - **S\_ISUID, S\_ISGID, S\_ISVTX**
  - **S\_IRWXU, S\_IRUSR, S\_IWUSR, S\_IXUSR**
  - **S\_IRWXG, S\_IRGRP, S\_IWGRP, S\_IXGRP**
  - **S\_IRWXO, S\_IROTH, S\_IWOTH, S\_IXOTH**

# Esempio

```
int main(void){
    struct stat statbuf;

    /* Pone a "uno" il bit set-group ID ed a "zero" il permesso di
    esecuzione per il gruppo */

    if (stat("foo", &statbuf) < 0)
        { printf("stat error for foo"); exit(1)}
    if (chmod("foo", (statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0){
        printf("chmod error for foo");

    /* Pone le protezioni a "rw-r--r--" */
    if (chmod("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) < 0)
        printf("chmod error for bar");
    return 0;
}
```

# chown

```
#include <sys/types.h>
#include <unistd.h>
int  chown(const char *path, uid_t owner, gid_t group);
int  fchown(int fd,          uid_t owner, gid_t group);
int  lchown(const char *path, uid_t owner, gid_t group);
```

- Modificano il campo **st\_uid** ed **st\_gid** del file indicato dal pathname (chown e lchown) o dal file descriptor (fchown)
- Se il parametro “owner” o “group” e' uguali a -1, il campo corrispondente non viene modificato
- In molti sistemi, solo un processo del superuser puo' modificare il campo **st\_uid**
- Un processo puo' modificare il gruppo se
  - (a) e' owner del file e
  - (b) il parametro group e' uguale all'effective GID del processo o ad uno dei gruppi “alternativi”

# File size

- Il campo **st\_size** della struttura struct contiene la dimensione in byte del file
  - Ha senso solo per file regolari, directory e link simbolici
- Il campo **st\_blksize** contiene la dimensione “ottimale” del blocco per operazioni di I/O
  - Ottimizzano le performance di accesso
- Il campo **st\_blocks** indica il numero di blocchi allocati al file

# File truncation

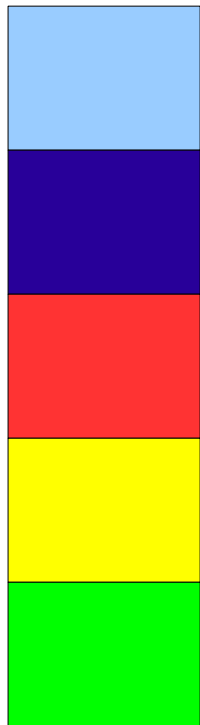
```
#include <unistd.h>
#include <sys/types.h>
int truncate(const char *path, off_t length);
int ftruncate(int fd, off_t length);
```

Restituiscono: 0 se Ok, -1 su errore

Queste funzioni “troncano” il file dopo “length” bytes.

- Se la dimensione e' maggiore del parametro length, i dati dal byte length+1 non sono piu' accessibili
- Se la dimensione e' minore di length, il comportamento e' system dependent
  - Nei sistemi linux, la dimensione del file viene aumentata e la parte in eccesso contiene “zeri”.

# Unix File system



“Header”: Boot block, superblock, Cylinder group info

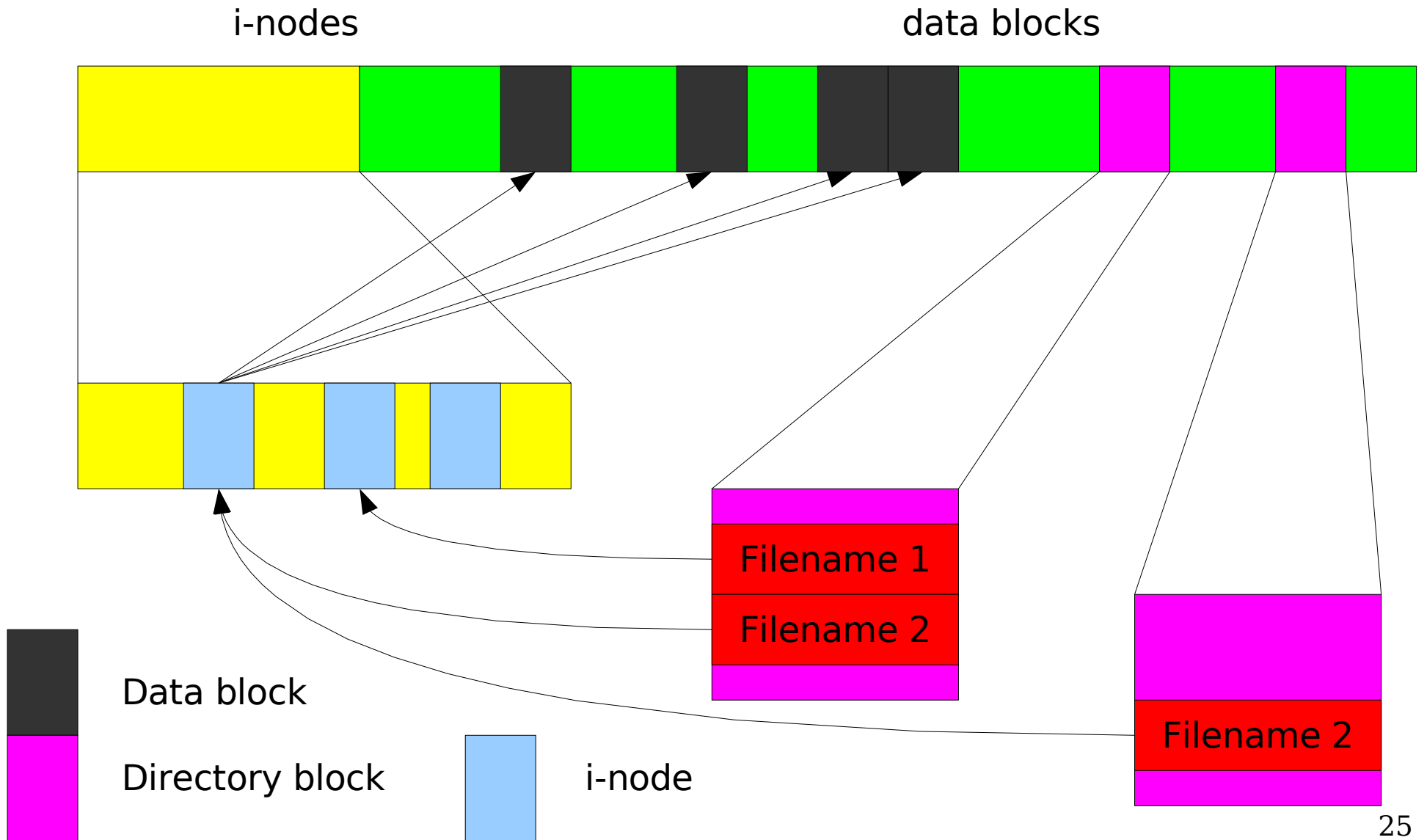
i-node bitmap

data block bitmap

i-node array

data block array

# Unix File system



# Unix File system

- La maggior parte delle informazioni della struttura stat sono contenute nell'i-node
- Il nome del file e l'i-node number (**st\_ino**) sono contenute nel directory block che contiene il file
- Il campo **st\_nlink** contiene il numero di link al file
  - Il file e' cancellato solo se questo campo ha valore 0.
- Per le directory il link count e' almeno 2
  - Un puntatore dalla directory “padre”
  - Un puntatore dalla directory “.”

# link/unlink

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

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

Restituiscono: 0 se OK, -1 su errore

- link: Crea un (hard) link tra oldpath (file esistente) e newpath.
  - Molti sistemi consentono la creazione di hard link alle directory solo ai processi con UID 0.
- unlink: Rimuove dalla directory il riferimento al file ed elimina il file se il campo **st\_nlink** diventa 0.
  - necessari permessi di scrittura e “ricerca” sulla directory o, se lo sticky bit e' settato, essere proprietari della directory o del file.
  - Nota: Il kernel elimina il file solo se non vi sono altri processi che lo usano

# rename

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

Restituisce: 0 se OK, -1 su errore

Rinomina il file indicato da oldpath come newpath.

- Se oldpath NON e' una directory, newpath NON puo' essere una directory. Se newpath esiste, viene cancellato.
- Se oldpath e' una directory
  - newpath deve essere una directory vuota
  - newpath non puo' contenere oldpath come prefisso
    - Non e' possibile rinominare /usr come /usr/bin
- Se oldpath o newpath e' un link, viene processato il link (non il file puntato dal link)

# Link simbolici

- I link simbolici consentono di:
  - Creare link tra entita' su filesystem diversi
  - La creazione di link a directory da parte di utenti.
- Directory entry:
  - Per un hard link contiene il numero di i-node del file “puntato”
  - Per un symbolic link contiene il puntatore ad un “data block” che contiene il nome del file puntato
- E' necessario controllare sempre se la funzione “segue” il link
  - e.g., “rename” di un symbolic link NON segue il link

# Link simbolici

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

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

```
int readlink(const char *path, char *buf, size_t bufsiz);
```

- symlink crea un link simbolico tra oldpath e newpath.
  - Non e' richiesto che oldpath esista.
- readlink: legge il "nome del file" a cui il link punta.