

# File e Directory

Laboratorio Sistemi Operativi

Giuseppe Salvi

Email: [giuseppe.salvi@uniparthenope.it](mailto:giuseppe.salvi@uniparthenope.it)

# Tipi di File

- La maggior parte dei file in Unix sono di due tipi: regolari e directory. Esistono anche tipi di file aggiuntivi
- I tipi di file sono:
  - **Regolari** (il tipo più comune di file contenente dati in una qualche forma)
    - Per il kernel non c'è distinzione tra dati testo o binari
  - **Directory** (contiene nomi di altri file e puntatori alle informazioni su tali file)
  - **File speciali a blocco**
    - sono usati per rappresentare dispositivi che consistono in un insieme di blocchi a indirizzamento casuale (dischi)
  - **File speciali a caratteri**
    - sono usati per rappresentare dispositivi che costituiscono flussi di caratteri (terminali, stampanti e interfacce di rete)
  - **FIFO**
    - usato per la comunicazione tra processi
  - **Socket**
    - tipo di file usato per la comunicazione su rete tra processi
  - **Link simbolici**
- Come possiamo ottenere queste informazioni?

# System call stat, fstat e lstat

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

```
int stat (const char *path, struct stat *buf);  
int fstat (int filedes, struct stat *buf);  
int lstat (const char *path, struct stat *buf);
```

- **stat** ritorna informazioni sul file specificato da **path**
- **fstat** ritorna informazioni sul file aperto sul descrittore **filedes**
- **lstat** ritorna informazioni sul link simbolico, non sul file puntato da esso
- Il secondo argomento è un puntatore ad una struttura. Le funzioni riempiono la struttura puntata da **buf**

# System call stat, fstat e lstat (cont.)

- Queste funzioni ritornano informazioni sul file
- Non è necessario avere permessi di lettura sul file per accedere a queste informazioni
- I permessi necessari sono di ricerca su tutte le directory nominate nel path

# La struttura stat

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

# Tipi di file

- L'informazione sul tipo di file si trova nel campo `st_mode` della struttura `stat`
- Per determinare il tipo di file si utilizzano le seguenti macro, definite in `<sys/stat.h>`. L'argomento di ciascuna macro è il campo `st_mode`

<code>S_ISLNK()</code>	symbolic link
<code>S_ISREG()</code>	regular file
<code>S_ISDIR()</code>	directory
<code>S_ISCHR()</code>	character device
<code>S_ISBLK()</code>	block device
<code>S_ISFIFO()</code>	FIFO
<code>S_ISSOCK()</code>	socket

# Esempio

```
#include <sys/types.h>
#include <sys/stat.h>
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) {
            printf("lstat error\n");
            continue;
        }
        if (S_ISREG(buf.st_mode)) ptr = "regular";
        ...
        else if (S_ISBLK(buf.st_mode)) ptr = "block special";

        else ptr = "*** unknown mode ***";
        printf("%s\n", ptr);
    }
    exit(0);}
}
```

# Esempio

```
$ ./a.out /etc/passwd /etc /dev/initctl /dev/log  
/dev/tty  
/etc/passwd: regular  
/etc: directory  
/dev/initctl: fifo  
/dev/log: socket  
/dev/tty: character special
```

# User ID e Group ID

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

# Campi `st_uid` e `st_gid`

- Ogni **file** ha un proprietario ed un gruppo che lo possiede
  - Tali informazioni si trovano in `st_uid` e `st_gid` di `stat`
- A ciascun **processo** vengono associati i seguenti identificativi:
  - `real user ID` e `real group ID`, identificano l'utente
  - `effective user ID`, `effective group ID`, e `supplementary group ID` determinano i permessi di accesso ai file
  - `saved set-user-ID` e `saved set-group-ID` contengono copia dell'`effective user ID` e `effective group ID` quando un programma è in esecuzione
- Hanno un ruolo fondamentale per i file eseguibili
- Normalmente, l'`effective user ID` coincide con il `real user ID` e l'`effective group ID` coincide con il `real group ID`

# Set user-ID

- Un programma è eseguito con i permessi di chi lo manda in esecuzione, non di chi lo possiede
- Si può inizializzare un flag in `st_mode` in modo che, quando un determinato file di programma viene eseguito, l'**effective user ID** del processo sia quello di chi possiede il file
- Tale flag è detto **set-user-ID**
  - potrebbe essere necessario che un processo abbia (in qualche momento) diritti maggiori di chi esegue il programma
  - Esempio: comando **passwd** (programma per cui set-user-ID è on)
    - Eseguito da chiunque
    - Deve avere la possibilità di cambiare il file `/etc/passwd`

# Set group-ID

- Analogamente è possibile mandare in esecuzione un processo con un effective group ID uguale al group ID del file eseguibile
  - Il relativo flag è detto **set-group-ID**
- Se il **set-group-ID** è applicato ad una directory, i file creati in quella directory ereditano il group ID dalla directory, non dall'effective group ID del processo che li ha creati

S_ISUID	4000	set UID bit
S_ISGID	2000	set GID bit
S_ISVTX	1000	sticky bit

# Set UID e Set GID: esempio

- Sia **exe** il nome di un file eseguibile. Siano **utente1** e **gruppo1** i valori di UID e GID dell'utente che lancia **exe**, producendo un processo **P**, che nel corso del proprio operato cercherà di accedere ad un file che chiameremo **info**
  - Il **real user ID** di **P** è l' **UID** dell'utente che lo ha generato (**utente1**). Analogamente, il **real group ID** di **P** è il **GID** dell'utente che lo ha generato (**gruppo1**)
  - L'**effective user ID** e l'**effective group ID** di **P** dipendono dai due bit speciali, **set user id** e **set group id**, associati all'eseguibile **exe**
  - Se **set user id** è attivo, l'**effective user id** di **P** sarà uguale all'**UID del proprietario** del file eseguibile; in caso contrario l'**effective user id** di **P** sarà uguale al suo **real user id** (cioè utente 1). Stesso discorso vale per **set group id**

# Set UID e Set GID: esempio (cont.)

- Vediamo come questi quattro identificatori sono utilizzati nel momento in cui il processo **P** cerca di accedere al file **info** (ricordiamo che **info** avrà associato un proprietario e un gruppo di utenti). Valgono le seguenti regole (nell'ordine elencato):
  - Se l'**effective user id** di **P** coincide con il proprietario di **info**, il processo acquisisce i **diritti di accesso del proprietario** di **info**
  - Altrimenti, se l'**effective group id** di **P** e il gruppo di **info** coincidono, **P** acquisisce i **diritti di accesso del gruppo** di utenti associati ad **info**
  - Se nessuna delle due precedenti condizioni è valida, valgono le normali triple di diritti di accesso: l'accesso sarà consentito o meno a seconda della categoria di utenti nella quale ricadono **real user id** e **real group id** del processo **P**

# Permessi di accesso ai file

- Complessivamente, sono dedicati 12 bit per i permessi



# Permessi di accesso ai file

- Come si è visto, `st_mode` contiene l'informazione relativa al tipo di file (regular file, directory,...)
- Il valore di `st_mode` codifica anche i bit di permesso di accesso ai file (`<sys/stat.h>`)

<code>S_IRUSR</code>	<code>0400</code>	owner read
<code>S_IWUSR</code>	<code>0200</code>	owner write
<code>S_IXUSR</code>	<code>0100</code>	owner execute
<code>S_IRGRP</code>	<code>0040</code>	group read
<code>S_IWGRP</code>	<code>0020</code>	group write
<code>S_IXGRP</code>	<code>0010</code>	group execute
<code>S_IROTH</code>	<code>0004</code>	others read
<code>S_IWOTH</code>	<code>0002</code>	others write
<code>S_IXOTH</code>	<code>0001</code>	others execute

# Sticky bit

- Sticky bit: permette di richiedere al kernel che l'immagine del **segmento di testo** di un processo resti allocata nell'area di swap anche dopo la sua terminazione (utile per programmi frequentemente utilizzati, es. vi)
- Se usato per le directory lo sticky bit (`S_ISVTX`)
  - I file nella directory possono essere **rinominati** o **cancellati** solo se l'utente ha i permessi di scrittura sulla directory e se vale una delle seguenti
    - è il proprietario del file
    - è il proprietario della directory
    - è il superutente
- Usato per directory per cui un qualsiasi utente può creare file
  - Gli utenti non dovrebbero avere la possibilità di cancellare o rinominare file di proprietà altrui

# Permessi di accesso ai file

- Le tre categorie read (r), write (w) e execute (x) sono usate da varie funzioni in maniera differente
  - Quando si vuole aprire un qualsiasi tipo di file mediante il nome, è necessario avere permessi di esecuzione (x) in ciascuna directory citata nel nome (inclusa la directory corrente)
    - Ciò motiva il fatto che il bit del permesso di esecuzione per le directory è anche chiamato bit di ricerca
    - Esempio: per aprire `/usr/include/stdio.h` dobbiamo avere i permessi di esecuzione nella directory `/`, nella directory `/usr` e nella directory `/usr/include`. E' necessario avere anche i permessi sul file stesso a seconda di come cerchiamo di aprirlo (sola lettura, scrittura, etc.)
    - I permessi di lettura ed esecuzione su una directory hanno significati differenti: **leggere la directory** ci consente di elencarne il contenuto; **l'esecuzione** ci consente di passare attraverso la directory quando è una componente del pathname a cui cerchiamo di accedere

# Permessi di accesso ai file

- Le tre categorie read, write e execute sono usate da diverse funzioni in maniera differente
  - I permessi di lettura determinano se possiamo aprire un file esistente per leggerlo (O\_RDONLY e O\_RDWR)
  - I permessi di scrittura determinano se possiamo aprire un file per la scrittura (O\_WRONLY e O\_RDWR)
  - E' necessario avere permessi di scrittura su di un file per specificare il flag O\_TRUNC nella funzione open
  - Non è possibile creare un file in una directory a meno che non si hanno i permessi di scrittura ed esecuzione sulla directory
  - Per cancellare un file esistente è necessario avere i permessi di scrittura ed esecuzione nella directory che contiene il file. Non è necessario avere permessi di lettura o scrittura sul file stesso
  - E' necessario avere i permessi di esecuzione su di un file se lo vogliamo eseguire usando una delle funzioni della famiglia exec. Inoltre, il file deve essere un file regolare

# Test di accesso ai file del kernel

- I permessi di accesso quando un processo apre, crea o cancella un file dipendono dal proprietario del file (`st_uid`, `st_gid`), gli effective ID del processo (`effective user ID` e `effective group ID`), e il supplementary group ID del processo
  - Gli ID del proprietario sono una prerogativa dei file
  - I due effective ID ed il supplementary group ID sono una prerogativa del processo
- I passi eseguiti in sequenza sono
  - Se l'`effective user ID` del processo è 0 (il superutente), l'accesso è consentito
  - Se l'`effective user ID` del processo coincide con il proprietario del file (ovvero, il processo è il proprietario), l'accesso è consentito se l'utente ha i relativi bit di permesso impostati, altrimenti l'accesso è negato
  - Se l'`effective group ID` del processo o uno dei `supplementary group ID` del processo coincide con il `group ID` del file, l'accesso è consentito se i permessi di accesso al gruppo sono impostati, altrimenti l'accesso è negato
  - Se gli altri permessi di accesso appropriati sono impostati, l'accesso è consentito altrimenti è negato

# Chiamata di sistema access

```
#include<unistd.h>
int access (const char *pathname, int mode)
```

- access effettua il test di accessibilità di un file sulla base del *real user ID* a del *real group ID*
  - Sostituisce *effective* con *real* nei quattro passi visti nella slide precedente

mode è il valore F\_OK per verificare se il file esiste oppure un OR bit a bit con le costanti R\_OK, W\_OK e X\_OK
- Restituisce 0 se Ok, altrimenti -1

# Esempio

```
#include<fcntl.h>
int main(int argc, char*argv[]){
    if (argc!=2){
        printf("usage: a.out <pathname>");
        exit(-1);
    }
    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);
}
```

# Esempio (cont.)

```
$ls -l a.out
-rwxr-xr-x 1 giusal giusal 8632 Mar 18 14:00 a.out
$ ./a.out a.out
Read access OK
Open for reading OK
$ls -l /etc/shadow
-rw-r----- 1 root shadow 1376 mar 16 12:36 /etc/shadow
$ ./a.out /etc/shadow
Access error for /etc/shadow
Open error for /etc/shadow
$ su
Password:
# chown root a.out
# chmod u+s a.out
# ls -l a.out
-rwsrwxr-x 1 root shadow 1376 mar 18 14:07 a.out
#exit
$ ./a.out /etc/shadow
Access error for /etc/shadow: Permessione denied
Open for reading OK
```

# Proprietà dei nuovi file e directory

- Lo user ID di un nuovo file è impostato all'effective user ID del processo
- Per determinare il group ID, POSIX.1 consente alla implementazione di scegliere tra le due opzioni
  - Il group ID di un nuovo file può essere l'effective group ID del processo
  - Il group ID del nuovo file può essere il group ID della directory in cui il file viene creato
- FreeBSD impiega sempre la seconda opzione
- Linux (filesystem ext2 e ext3) consente di scegliere l'una o l'altra sulla base di un flag impostato con mount
  - Linux (4.2.22 con l'apposita opzione per mount) e Solaris 9 scelgono l'una o l'altra sulla base dell'impostazione del bit set-group-ID per la directory in cui creare il file

# Permessi e creazione dei file

```
#include <sys/stat.h>  
mode_t umask(mode_t mask);
```

- La system call umask viene utilizzata per assegnare ad un processo la modalità di creazione di un file
- L'argomento mask è formato da un OR bit a bit delle nove costanti di permesso di accesso ai file
- La funzione ritorna il valore precedente della maschera di creazione dei file

# La system call umask

- La maschera della modalità di creazione di un file è usata ogniqualvolta il processo crea un nuovo file o una nuova directory
- I permessi di un file creato, dato un valore della maschera, sono calcolati usando la seguente operazione bit a bit
  - AND bit a bit tra il complemento di mask e la modalità di accesso specificata in **creat** o **open**
- Esempio: se il valore di default di umask viene inizializzato a 022, un nuovo file creato con permessi 666 avrà
  - $666 \& \sim 022 = 644 = r w - r - - r - -$

# Esempio

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

int main(void) {

    umask(0);
    if (creat("foo", S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH
        | S_IWOTH) < 0)
        err_sys("creat error for foo");

    umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
    if (creat("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
        S_IROTH | S_IWOTH) < 0)
        err_sys("creat error for bar");
    exit(0);
}
```

# System call chmod e fchmod

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

- Queste funzioni permettono di cambiare i permessi di accesso ad un file
  - Restituiscono 0 se OK, -1 in caso di errore
- Per cambiare i bit di permesso di un file, l'**effective user ID** del processo deve essere uguale all' **ID del proprietario**, oppure il processo deve avere i diritti del superutente

# Esempio

```
#include <sys/types.h>
#include <sys/stat.h>
#include "apue.h"
int main(void)
{
    struct stat statbuf;
    /* modo assoluto a "rw-r--r--" */
    if (chmod("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) <
        0)
        err_sys("chmod error for bar");
    /* imposta set-group-ID disattiva group-execute */
    if (stat("foo", &statbuf) < 0)
        err_sys("stat error for foo");
    if (chmod("foo", (statbuf.st_mode & ~S_IXGRP) | S_ISGID) <
        0)
        err_sys("chmod error for foo");

    exit(0);
}
```

# System call chown, fchown e lchown

```
#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);
```

- Tali funzioni permettono di cambiare lo user ID ed il group ID di un file
  - Ritornano 0, se OK, -1 in caso di errore
- Se l'argomento owner o group è -1, l'ID corrispondente è lasciato inalterato
- Solo un processo superutente può modificarne il proprietario
- Un processo non superutente, proprietario del file (**effective user ID e real user ID coincidono**) può solo modificarne il gruppo con uno tra quelli **supplementari** a cui appartiene

# Dimensione dei file

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

# Dimensione dei file

- Il campo `st_size` di `stat` contiene la dimensione in byte del file. Ha senso solo per file regolari, directory e link simbolici
  - Solaris definisce anche la dimensione del file di una pipe come il numero di byte disponibili per lettura dalla pipe
- Se presenti, `st_blksize` e `st_blocks` si riferiscono, rispettivamente, al migliore fattore di blocco per eseguire operazioni di I/O sul file e al numero di blocchi da 512 byte allocati per il file
- La libreria standard del C utilizza tale fattore di blocco per eseguire le operazioni su file

# Troncamento

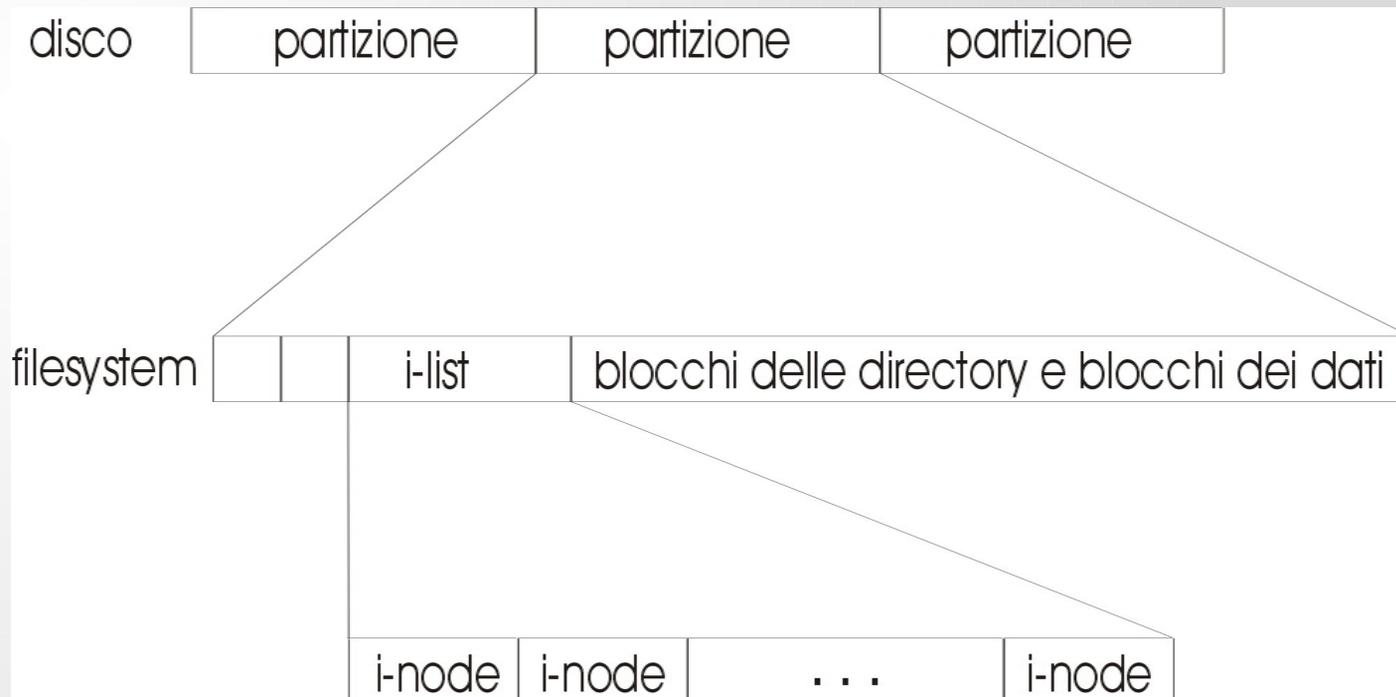
```
#include <unistd.h>
int truncate (const char *path, off_t length);
int ftruncate (int fd, off_t length);
```

- Queste system call troncano un file esistente a **length** byte
  - Ritornano 0 se OK, -1 in caso di errore
- Se il file ha una dimensione maggiore di length, i dati oltre length non sono più accessibili

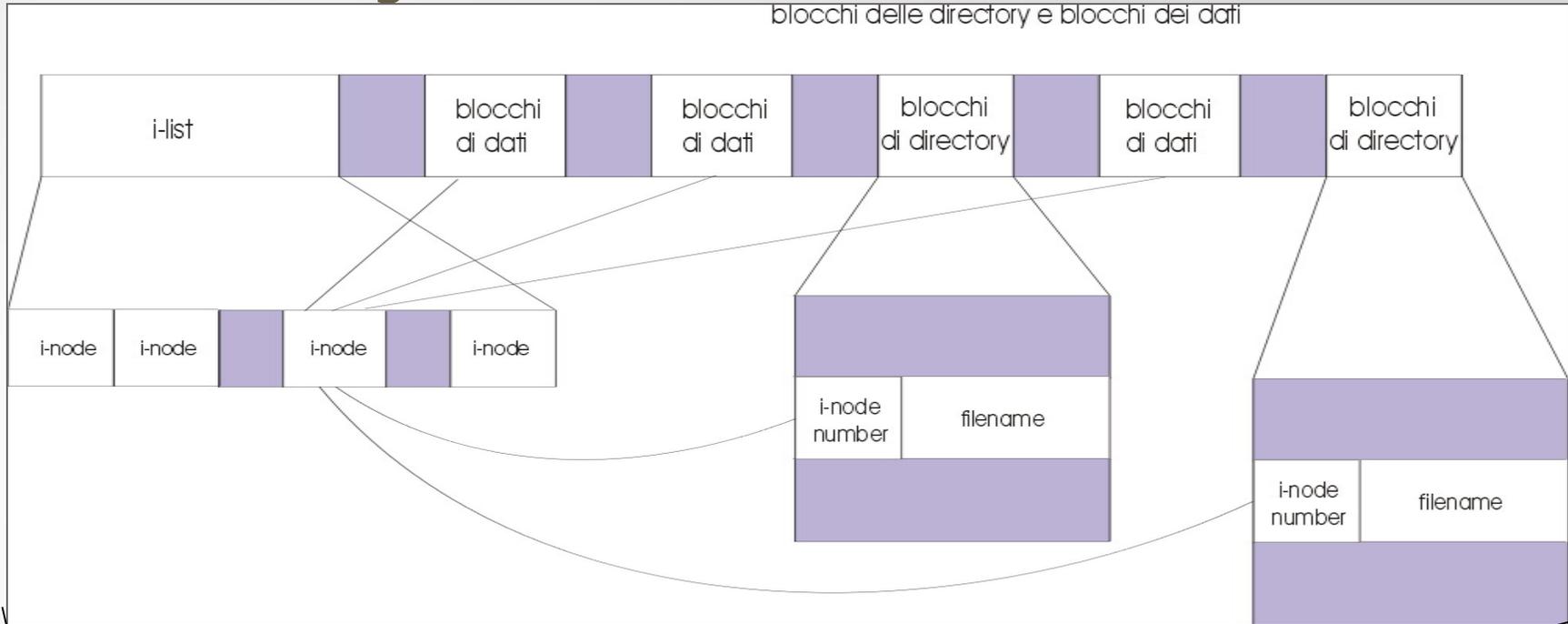
# File system

# File system

- Consideriamo il file system di UNIX (UFS) derivato da BSD
  - Possiamo pensare un HD suddiviso in una o più partizioni
    - Ogni partizione può contenere un file system



# File system



# I-node

- Index-Node è una struttura di controllo associata ad ogni file
  - molti nomi dei file possono essere associati con lo stesso i-node

<b>File Mode</b>	16-bit flag that stores access and execution permissions associated with the file.
	12-14 File type (regular, directory, character or block special, FIFO pipe)
	9-11 Execution flags
	8 Owner read permission
	7 Owner write permission
	6 Owner execute permission
	5 Group read permission
	4 Group write permission
	3 Group execute permission
	2 Other read permission
	1 Other write permission
	0 Other execute permission
<b>Link Count</b>	Number of directory references to this inode
<b>Owner ID</b>	Individual owner of file
<b>Group ID</b>	Group owner associated with this file
<b>File Size</b>	Number of bytes in file
<b>File Addresses</b>	39 bytes of address information
<b>Last Accessed</b>	Time of last file access
<b>Last Modified</b>	Time of last file modification
<b>Inode Modified</b>	Time of last inode modification

# Link hard e simbolici

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

# System call link

- E' possibile far puntare più directory all'i-node di un file
- La maniera per creare un hard link ad un file esistente è quella di usare la funzione link

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

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

Ritorna 0 se OK, -1 in caso di errore

- Se newpath già esiste è ritornato un errore
- Se un'implementazione supporta la creazione di hard link su directory, il privilegio è ristretto al solo superutente

# System call unlink

- Per rimuovere un elemento dalla tabella della directory si utilizza la funzione unlink

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

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

Ritorna 0 se OK, -1 in caso di errore

- La funzione decrementa il numero di link del file puntato da pathname. Il file risulta ancora accessibile, se il numero di link è non nullo

# System call unlink (cont.)

- Quando il numero di link del file è 0, il contenuto del file può essere cancellato
- Ciò non accade se un processo ha il file aperto
- Quando il file viene chiuso, il kernel conta il numero di processi che hanno aperto il file: se questo è zero ed il numero di link del file è zero, allora il file è cancellato
- E' necessario avere i permessi di scrittura ed esecuzione nella directory contenente il file

# System call symlink e readlink

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

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

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

- symlink viene utilizzata per creare un link simbolico. Non è necessario che oldpath esista
  - Restituisce 0 se OK, -1 in caso di errore
- Per leggere un link simbolico, è necessario utilizzare readlink, che apre il link, legge il contenuto e lo chiude
  - Restituisce il numero di byte letti se OK, -1 in caso di errore
- Il contenuto del link è posto in buf, senza carattere di terminazione

# System call mkdir e rmdir

```
#include <sys/stat.h>  
int mkdir (const char *pathname, mode_t mode);
```

```
#include <unistd.h>  
int rmdir (const char *pathname);
```

- Queste funzioni permettono di creare directory e rimuoverle (se vuote)
  - Ritornano 0 se OK, -1 in caso di errore

# Letture delle directory

- Una directory può essere letta da chiunque abbia i permessi di accesso per lettura
- I permessi di accesso e scrittura per una directory determinano se si possono creare nuovi file nella directory e se si possono cancellare
- I permessi non specificano se si può scrivere sui file contenuti nella directory stessa

# Lettura delle directory (funzioni libreria)

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

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

Ritorna un puntatore se OK, NULL in caso di errore

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

Ritorna un puntatore se OK, NULL alla fine o in caso di errore

```
    struct dirent {  
        ino_t d_ino;  
        char d_name[NAME_MAX + 1];  
    };
```

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

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

- Il puntatore ad una struttura di tipo **DIR** ritornato da **opendir** è utilizzato dalle altre funzioni

# Esempio: elencare i file di una directory

```
#include <sys/types.h>
#include <dirent.h>
int main(int argc, char *argv[ ])
{
    DIR *dp;
    struct dirent *dirp;
    if (argc != 2){
        printf("a single argument (the directory name) is required");
        exit(-1);
    }
    if ( (dp = opendir(argv[1])) == NULL){
        printf("can't open %s", argv[1]);
        exit(-1);}
    while ( (dirp = readdir(dp)) != NULL)
        printf("%s\n", dirp->d_name);
    closedir(dp);
    exit(0);
}
```

# System call chdir e funzione getcwd

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

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

Ritorna 0 se OK, -1 in caso di errore

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

Ritorna puntatore a buf se OK,

NULL in caso di errore

- Ciascun processo è dotato di una directory di lavoro corrente da cui partono tutti i path relativi
- Per sapere qual è la directory corrente si usa `getcwd`
- Per cambiare la directory di lavoro corrente di un processo si usa la funzione `chdir`

# Esempio

- La directory di lavoro corrente è un attributo di un processo
  - I processi che invocano il processo che esegue chdir non sono influenzati dal cambio di directory di quest'ultimo

```
#include      <unistd.h>
#include      <stdio.h>
int main(int argc, char *argv[])
{
    if (chdir("/tmp") < 0){
        perror("chdir fallito");
        exit(-1);
    } else
        printf("chdir a /tmp avvenuto\n");
        exit(0);
}
```

# Esempio (2)

```
$ pwd  
/users/studente  
$ a.out  
chdir a /tmp avvenuto  
$ pwd  
/users/studente  
$
```

- La directory di lavoro corrente per la shell non è cambiata
  - E' un effetto di come la shell esegue i programmi
  - Ogni programma viene eseguito in un processo separato
    - La directory di lavoro corrente della shell non è influenzata dalla chiamata a chdir nel programma

# Esempio getcwd

```
#include "apue.h"
int main(void)
{
char *ptr;
int size;

if (chdir("/home/giusal/bin") < 0)
err_sys("chdir fallito");

ptr = path_alloc(&size); /* funzione in apue.h */
if (getcwd(ptr, size) == NULL)
err_sys("getcwd fallita");

printf("cwd = %s\n", ptr);
exit(0);
}
```

# Esempio getcwd (2)

```
$pwd  
/home/giusal  
$ ln -s /usr/bin ./bin  
$ ./a.out  
cwd = /usr/bin  
$
```

# Esercizio

- Creare un link simbolico con il comando `ln -s`. Scrivere un programma che selezioni il link nella directory e stampi a video il nome del file.

# Soluzione

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <dirent.h>
int main()
{
    char c, cwd[100];
    int fd;
    struct stat buf;
    struct dirent *drn;
    DIR *dp;
    getcwd(cwd, sizeof(cwd));

    if((dp=opendir(cwd))==NULL){ // 'carica' la struttura associata alla directory
        printf("opendir error\n");
        exit(-1);
    }

    while((drn=readdir(dp))!=NULL){ // legge il contenuto della dir
        if(lstat(drn->d_name,&buf) <0)
            printf("Errore lstat su %s\n",drn->d_name);
        if(S_ISLNK(buf.st_mode))
            printf("Trovato il link %s\n",drn->d_name);
    }

    closedir(dp);
    exit(0);
}
```