Modulo 8
System call per la gestione dei file

Laboratorio di Sistemi Operativi I
Anno Accademico 2008-2009

Principali system call per la gestione dei file

Nome	Significato
`open`	apre un file in lettura e/o scrittura o crea un nuovo file
`creat`	crea un nuovo file
`close`	chiude un file precedentemente aperto
`read`	legge da un file
`write`	scrive su un file
`lseek`	sposta il puntatore di lettura/scrittura ad un byte specificato
`unlink`	rimuove un file
`ioctl`	permette la manipolazione di parametri di un device
`dup`	duplica un file descriptor
`fcntl`	controlla gli attributi associati ad un file e permette il file locking
`stat`	ritorna informazioni su un file

Note introduttive (I)

- Le system call descritte in questo modulo hanno un equivalente nella libreria di I/O standard C (prototipi in **`<stdio.h>`**)
- E' importante conoscere il concetto di operazione atomica, dato che i file possono essere condivisi tra piu' processi
- Un file per essere usato deve essere aperto (**`open`**)
- Per il kernel, ad ogni file aperto e' associato un *file descriptor*
- L'operazione **`open`**:
  - localizza il file nel file system attraverso il suo pathname
  - copia in memoria il descrittore del file (*i-node*)
  - associa al file un intero non negativo (*file descriptor*), che verrà usato nelle operazioni di accesso al file, invece del pathname

Note introduttive (II)

- Quando non si ha più necessità di accedere al file, bisogna chiuderlo
- La **`close`** rende disponibile il file descriptor per ulteriori usi
- I file standard non devono essere aperti, perché sono già aperti dalla shell.
- Essi sono associati ai file descriptor:
  0 = standard input (stdin)
  1 = standard output (stdout)
  2 = standard error (stderr)
- In accordo allo standard POSIX.1, tali numeri dovrebbero essere sostituiti dalle relative costanti simboliche, definite in **`<unistd.h>`**:
  0 = **`STDIN_FILENO`**
  1 = **`STDOUT_FILENO`**
  2 = **`STDERR_FILENO`**

Condivisione di file (I)

- Linux supporta la condivisione dei file (***file sharing***) aperti tra processi differenti
- Per comprendere i meccanismi del file sharing è necessario conoscere le strutture dati che il kernel mantiene per i file aperti
- Ricordiamo che il kernel mantiene in memoria una ***process table*** con le informazioni di ogni processo
- Le strutture dati relative ai file aperti mantenute dal kernel sono:
  - ***file table***, i cui elementi sono le ***file table entry***
  - ***v-node table,*** i cui elementi sono i ***v-node***

Condivisione di file (II)

- Ad ogni processo e' riservata un'area dedicata nella ***process table (process table entry)***
- All'interno di ogni ***process table entry*** c'è una tabella, detta ***table of open file descriptors***, relativa ad ogni singolo processo (*process-wide*), che contiene i descrittori di file aperti per quel processo
- Sono associati a ciascun ***file descriptor*** nella ***table of open file descriptors***:
  - I flag del file descriptor
  - Un puntatore a una ***file table entry***

Condivisione di file (III)

- La ***file table*** e' una tabella dei file aperti da tutti i processi nel sistema (system-wide)
- Ogni entry della ***file table***, detta ***file table entry,*** contiene:
  - i flag di stato del file (read, write, append, sync, nonblocking, ...)
  - Il file offset corrente
  - Un puntatore alla entry corrispondente al file in questione nella ***v-node table***

Condivisione di file (IV)

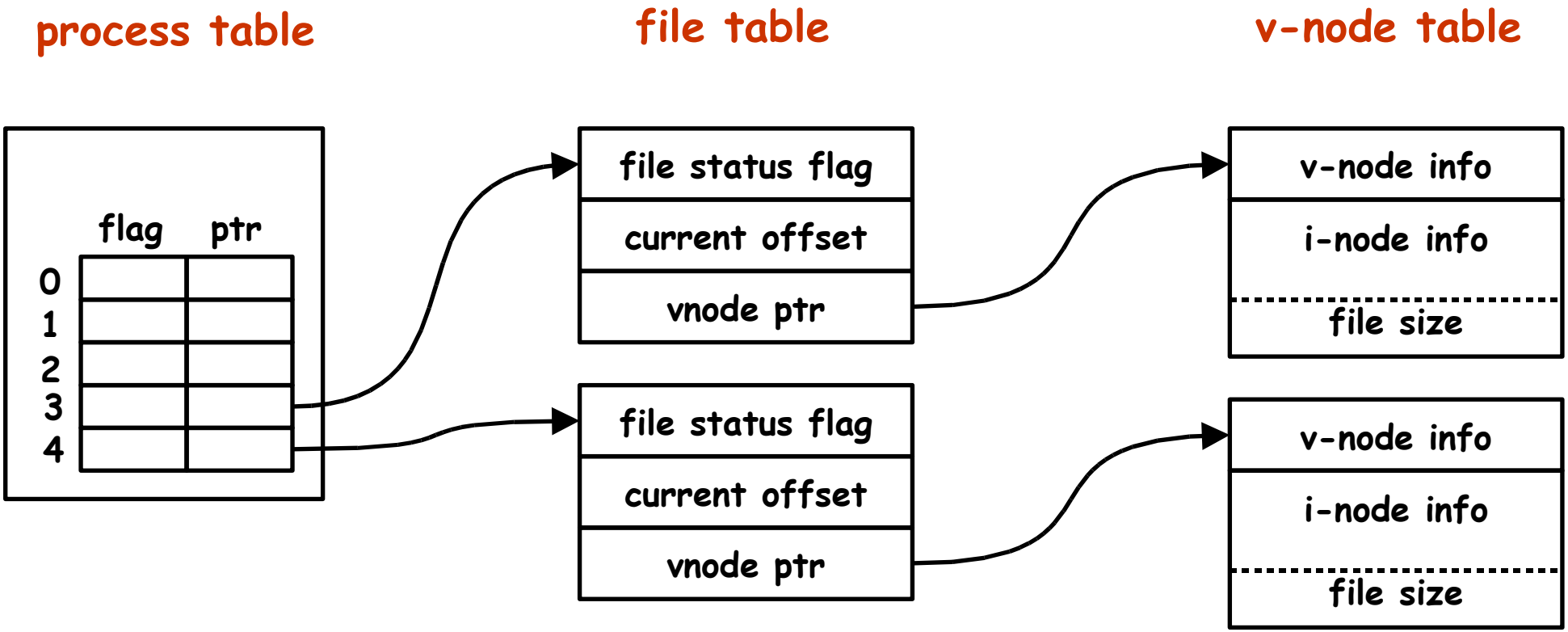
- La ***v-node table*** e' una tabella di ***v-node***, i quali contengono informazioni sul tipo del file e i puntatori alle funzioni che possono operare sul file
- Ogni entry della ***v-node table*** contiene:
  - Informazioni sul tipo di file
  - Puntatori alle funzioni che operano sul file
  - In molti casi, contengono l'i-node del file (o un puntatore ad esso). L'i-node contiene il proprietario del file, la dimensione, i puntatori ai blocchi dei dati del file, ...

Condivisione di file (V)

```
struct inode {
        struct hlist_node        i_hash;
        struct list_head         i_list;
        struct list_head         i_sb_list;
        struct list_head         i_dentry;
        unsigned long            i_ino;
        atomic_t                 i_count;
        umode_t                  i_mode;
        unsigned int             i_nlink;
        uid_t                    i_uid;
        gid_t                    i_gid;
        dev_t                    i_rdev;
        loff_t                   i_size;
        struct timespec          i_atime;
        struct timespec          i_mtime;
        struct timespec          i_ctime;
        unsigned int             i_blkbits;
        unsigned long            i_blksize;
        unsigned long            i_version;
        unsigned long            i_blocks;
        unsigned short           i_bytes;
        spinlock_t               i_lock;
        struct semaphore         i_sem;
        struct rw_semaphore      i_alloc_sem;
        struct inode_operations *i_op;
        struct file_operations  *i_fop;
        struct super_block      *i_sb;
        struct file_lock        *i_flock;
        struct address_space    *i_mapping;
        struct address_space     i_data;
#ifdef CONFIG_QUOTA
        struct dquot
    *i_dquot[MAXQUOTAS];
#endif
        struct list_head         i_devices;
        struct pipe_inode_info  *i_pipe;
        struct block_device     *i_bdev;
        struct cdev             *i_cdev;
        int                      i_cindex;
        __u32                    i_generation;
#ifdef CONFIG_DNOTIFY
        unsigned long            i_dnotify_mask;
        struct dnotify_struct   *i_dnotify;
#endif
        unsigned long            i_state;
        unsigned long            dirtied_when;

        unsigned int             i_flags;

        atomic_t                 i_writecount;
        void                    *i_security;
        union {
                void            *generic_ip;
        } u;
#ifdef __NEED_I_SIZE_ORDERED
        seqcount_t               i_size_seqcount;
#endif
};

// file: /usr/src/kernels/.../include/linux/fs.h
```

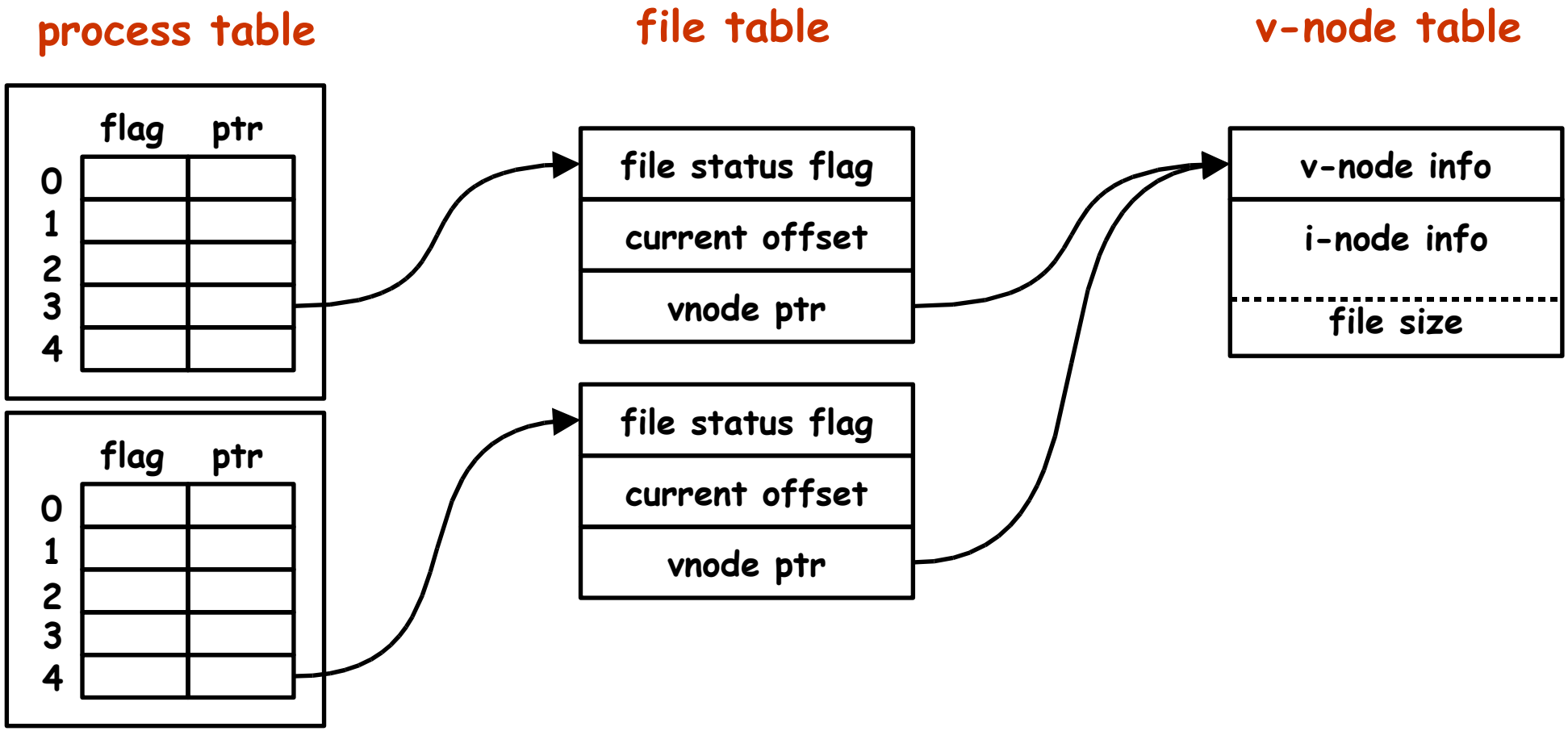
Condivisione di file (VI)

Esempio: le tre tabelle con un processo che apre due file distinti



Condivisione di file (VII)

Esempio: le tre tabelle con due processi che aprono lo stesso file



Condivisione di file (VIII)

- **NOTA**: due processi che aprono lo stesso file ottengono due distinte *file table entry*, quindi ogni processo ha il proprio *current offset* per il file
- Vista questa organizzazione, possiamo essere più specifici riguardo alle operazioni viste in precedenza:
  - alla conclusione di ogni **`write`**
    - il current offset nella file table entry viene incrementato
    - se il current offset è maggiore della dimensione del file nella v-node table entry, questa viene incrementata
  - se il file è aperto con il flag **`O_APPEND`**
    - un flag corrispondente è settato nella file table entry
    - ad ogni **`write`**, il current offset viene prima posto uguale alla dimensione del file nella v-node table entry
  - **`lseek`** modifica unicamente il current offset nella file table entry corrispondente

Condivisione di file (IX)

- E' possibile che più ***file descriptor*** siano associati a una singola ***file table entry***
  - tramite la funzione **`dup`**, all'interno dello stesso processo
  - tramite **`fork`**, tra due processi diversi
- E' interessante notare che esistono due tipi di flag:
  - alcuni sono associati al ***file descriptor***, e quindi sono particolari del processo
  - altri sono associati alla ***file table entry***, e quindi possono essere condivisi fra più processi
  - esiste la possibilità di modificare questi flag (funzione **`fcntl`**)
- E' importante notare che questo sistema di strutture dati condivise può portare a problemi di concorrenza

Apertura di file (I)

```
int open(const char *pathname, int flags);
```

- apre il file specificato da **`pathname`** (assoluto o relativo), secondo la modalità di accesso specificata in **`flags`**
- restituisce il file descriptor con il quale ci si riferirà al file successivamente (o -1 se errore)

- **Valori di `flags`**
  - **`O_RDONLY`** read-only (0)
  - **`O_WRONLY`** write-only (1)
  - **`O_RDWR`** read and write (2)
  - Solo una di queste costanti può essere utilizzata in **`flags`**
  - Altre costanti (che vanno aggiunte in OR ad una di queste tre) permettono di definire alcuni comportamenti particolari

Apertura di file (II)

- **Altri valori di `flags`:**
  - **`O_CREAT`** creazione di un nuovo file
  - **`O_APPEND`** scrittura alla fine del file
  - **`O_EXCL`** se il file esiste ed e' in OR con **`O_CREAT`**, ritorna un errore
  - **`O_TRUNC`** se il file esiste, viene svuotato
  - **`O_NONBLOCK`** imposta operazioni non-bloccanti per file speciali, quali pipe e FIFO
  - **`O_SYNC`** synchronous write, cioe' ogni write ritorna solo quando i dati sono effettivamente stati scritti sul dispositivo hardware
- Se si specifica **`O_CREAT`**, è necessario specificare un terzo argomento:
  `int open(const char *pathname, int flags, mode_t mode);`
- Per il significato e i valori di **`mode`** si veda prossima slide relativa a **`creat`**

Creazione di file

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

- crea un nuovo file regolare con **`pathname`** specificato, e lo apre in scrittura
- se il file esiste già, lo svuota (owner e **`mode`** restano invariati)
- **`mode`** specifica i permessi iniziali; l'owner del file coincide con l'effective user-id del processo
- **`mode`** codifica i permessi di accesso al file mediante un numero ottale (ad esempio **`0644 = rw-r--r--`**).
- le due system call ritornano il file descriptor, o -1 in caso di errore
- Equivalenze:

  `creat(pathname, mode);`

  ha lo stesso effetto di

  `open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);`

Chiusura di file

```
int close(int filedes);
```

- chiude il file associato al file descriptor **`filedes`**
- ritorna l'esito dell'operazione (0 o -1)

- **Nota:**
  - Quando un processo termina, tutti i suoi file rimasti aperti vengono comunque chiusi automaticamente dal kernel
  - Nonostante cio' e' SEMPRE buona norma fare in modo di chiudere esplicitamente i file aperti prima della terminazione del processo

Esempio

```
int fd;
...
fd=open(pathname, ...);
if (fd==-1)
{
   /*gestione errore*/
}
/* Il file è aperto correttamente */
read(fd, ...);
write(fd,...);
close(fd);
/* Il file è chiuso */
```

- **Nota:** Un file può essere aperto più volte contemporaneamente, e quindi avere più file descriptor associati.

Lettura di byte da file

```
ssize_t read(int filedes, void *buf, size_t nbyte);
```

- legge dal file identificato dal descriptor **`filedes`** una sequenza di **`nbyte`** byte a partire dal puntatore di lettura/scrittura e la salva nel buffer che inizia da **`buf`**
- aggiorna il valore del *current file offset*
- restituisce il numero di bytes **effettivamente** letti (che può essere uguale o minore di **`nbyte`**), o -1 se errore
- il numero di byte letti può risultare inferiore al numero di byte richiesti nei seguenti casi:
  - si è giunti alla fine di un file regolare
  - si sta leggendo da terminale, da una pipe o da uno stream proveniente dalla rete e non ci sono abbastanza byte in input

Scrittura di byte su file

```
ssize_t write(int filedes, const void *buf, size_t nbyte);
```

- scrive nel file identificato dal file descriptor **`filedes`** una sequenza di **`nbyte`** byte a partire dal puntatore di lettura/scrittura, leggendola dal buffer che inizia da **`buf`**
- aggiorna il valore del *current file offset*
- restituisce il numero di bytes **effettivamente** scritti, o -1 se errore (0 indica che nessun byte è stato scritto)
- una write completata senza errori non garantisce che i dati siano stati scritti effettivamente sul disco; il modo per essere sicuri è quello di invocare la **`fsync`** dopo che sono state effettuate le varie chiamate **`write`**

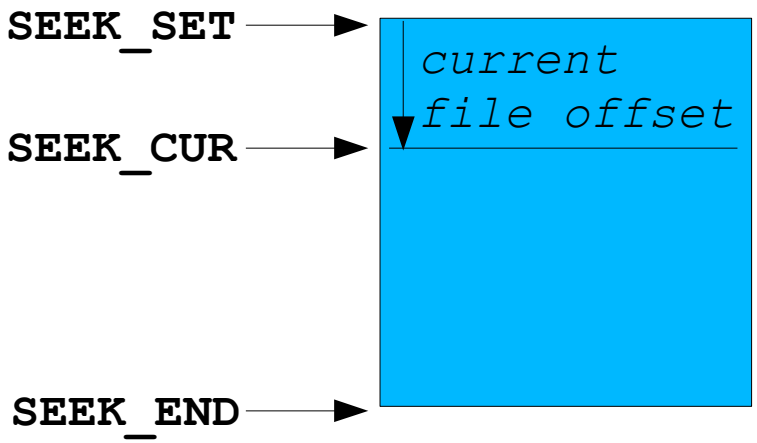
Current file offset

- Ad ogni file aperto e' associato un *current file offset*, che indica la posizione attuale del puntatore di lettura/scrittura all'interno del file
- L'offset è un valore non negativo che misura il numero di byte dall'inizio del file
- Le operazioni **`read`**/**`write`** leggono/scrivono dalla posizione attuale e incrementano il current file offset in avanti del numero di byte letti/scritti
- Quando viene aperto un file, il *current file offset* viene posizionato a 0, a meno che non sia stata specificata l'opzione **`O_APPEND`**, nel qual caso viene posizionato dopo l'ultimo byte del file

Spostamento all'interno di un file

```
off_t lseek(int filedes, off_t offset, int whence);
```

- sposta la posizione del puntatore di lettura/scrittura nel file **`filedes`** di **`offset`** byte a partire dalla posizione specificata dal parametro **`whence`**, che può assumere i valori:
  - **`SEEK_SET`** dall'inizio del file
  - **`SEEK_CUR`** dalla posizione corrente
  - **`SEEK_END`** dalla fine del file



- restituisce il *current file offset* dopo l'esecuzione di **`lseek`**, o -1 se errore
- **Nota:**
  - La **`lseek`** non effettua alcuna operazione di I/O

Offset validi e non validi

- Il parametro **`offset`** può essere negativo, cioè sono ammessi spostamenti all'indietro a partire dalla posizione indicata da **`whence`**.
- **Tentativi di spostamento prima dell'inizio del file generano un errore. Pero´ e´ possibile spostarsi oltre la fine del file**. Ovviamente non ci saranno dati da leggere in tale posizione e futuri accessi tramite la **`read`** ai byte compresi tra la vecchia fine del file e la nuova posizione daranno come risultato il carattere ASCII null (`'\0'`).
- **Esempio:**

```
off_t newpos;
...
newpos = lseek(fd, (off_t)-16, SEEK_END);
/* newpos punta a 16 byte prima della fine */
newpos = lseek(fd, (off_t)16, SEEK_END);
/* newpos punta a 16 byte dopo la fine */
```

"Buchi" nei file

- La dimensione di un file e il numero di blocchi occupati su disco possono non coincidere: questo avviene quando in un file c'è un "buco" creato da `lseek`

Esempio:

```
$ df --block-size=M .
Filesystem           1M-blocks      Used Available Use% Mounted on
/dev/sda6               38822M     5992M    30858M  17% /home
$ ./creatFileBigHole
$ ls -l --block-size=M filebighole
-rw-r----- 1 verola verola 1025M 2007-11-08 18:54 filebighole
$ df --block-size=M .
Filesystem           1M-blocks      Used Available Use% Mounted on
/dev/sda6               38822M     5992M    30858M  17% /home
$ stat filebighole
  File: `filebighole'
  Size: 1073741832      Blocks: 32         IO Block: 4096   regular file
```

Scrittura alla fine di un file

- Vi sono due modi per scrivere alla fine di un file:
  - usare **lseek** per spostarsi alla fine del file:

```
lseek(fildes, (off_t)0, SEEK_END);
write(fildes, buf, BUFSIZE);
```

  - usare **open** con il flag **O_APPEND**:

```
fildes = open("nomefile", O_WRONLY | O_APPEND);
write(fildes, buf, BUFSIZE);
```

Esercizi 1

- A) Scrivere un programma che:
  - legge una linea dallo standard input e la stampa sullo standard output
  - esce quando la linea di input e' vuota (solo Enter)
- B) Scrivere un programma che:
  - prende come argomento la grandezza di un blocco (in bytes o kbytes o Mbytes) e il nome di un file di input
  - genera il numero necessario di file di output, ognuno contenente in sequenza un blocco del file di input (divisione del file di input in blocchi di pari dimensione); i nomi dei file di output devono essere generati automaticamente a partire dal nome del file di input
- C) Scrivere un programma che:
  - gestisce una matrice memorizzata su file (creazione, stampa, inserimento/estrazione di elementi, ...)

Esercizi 2

- A) Scrivere un programma che:
  - crea un file il cui filename e' l'unico argomento del programma stesso
  - genera un processo child che esegue lo stesso codice del parent
  - il parent e il child scrivono indipendentemente 10 linee nel file aperto e poi chiudono il file descriptor

  Verificare come si compongono nel file le linee scritte dai 2 processi.
  Vengono sovrascritte o le une si aggiungono alle altre?
  Lanciare il programma sviluppato piu' volte e verificare se il contenuto del file subisce modifiche.

- B) Modificare il programma spostando l'apertura del file dopo la generazione del processo child, ma sempre nella parte comune di codice.

  Come differisce il contenuto del file rispetto al risultato del programma precedente?

Modifica del file descriptor

- La funzione **`fcntl`** permette di esercitare un certo grado di controllo su file già aperti. Ha tre forme di chiamata:

```
int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock *lock);
```

- **`fd`:** è il descrittore del file su cui operare
- **`cmd`:** è il comando da eseguire
- Il terzo argomento, quando presente, è il parametro del comando:
  - **`arg`**: un valore intero (caso generale)
  - **`lock`**: un puntatore (caso di record locking)

- **Comandi:**
  - duplicazione di file descriptor (**`F_DUPFD`**)
  - get/set file descriptor flag (**`F_GETFD, F_SETFD`**)
  - get/set file status flag (**`F_GETFL, F_SETFL`**)
  - get/set async. I/O ownership (**`F_GETOWN, F_SETOWN`**)
  - get/set record locks (**`F_GETLK, F_SETLK, F_SETLKW`**)

Funzione `fcntl` (I)

```
int fcntl(int fd, F_DUPFD, long arg);
```

- Duplica il file descriptor specificato da **`filedes`**
- Ritorna il nuovo file descriptor
- Il file descriptor scelto è uguale al valore più basso corrispondente ad un file descriptor non aperto e che sia maggiore o uguale a **`arg`**

Funzione `fcntl` (II)

```
int fcntl(int fd, F_GETFD);
```

- Ritorna i file descriptor flag associati a **`fd`**
- Attualmente è definito un solo file descriptor flag, **`FD_CLOEXEC`**:
  - se **`FD_CLOEXEC`** è true (cioe' un valore diverso da 0), il file descriptor viene chiuso eseguendo una **`exec`**

```
int fcntl(int fd, F_SETFD, long arg);
```

- Modifica i file descriptor flag associati a **`fd`**, utilizzando il terzo argomento come nuovo insieme di flag
- Attualmente imposta il flag *close-on-exec* al valore specificato dal bit **`FD_CLOEXEC`** di **`arg`**.

Funzione `fcntl` (III)

```
int fcntl(int fd, F_GETFL);
```

- Ritorna i file status flag associati a **`fd`**
- I file status flag sono quelli utilizzati nella funzione **`open`**

- Esiste la maschera **`O_ACCMODE`** (uguale a 3) che permette di isolare la modalità di accesso, cioè:

```
oflag = fcntl(int fd, F_GETFL) & O_ACCMODE;
```

può essere uguale a uno dei valori:

`O_RDONLY`, **`O_WRONLY`**, **`O_RDWR`**

- Per determinare gli altri flag, è possibile utilizzare le costanti definite (**`O_APPEND`**, **`O_NONBLOCK`**, **`O_SYNC`**)

Funzione `fcntl` (IV)

```
int fcntl(int fd, F_SETFL, long arg);
```

- Modifica i file status flag associati a **`fd`** con il valore specificato in **`arg`**
- I soli valori che possono essere modificati sono **`O_APPEND`**, **`O_NONBLOCK`**, **`O_ASYNC`** e **`O_DIRECT`**;
  l'access mode deve rimanere inalterato

Esempio 1 d'uso di `fcntl` (I)

```
#include <fcntl.h>

int filestatus(int filedes)
{
    int arg1;
    if((arg1 = fcntl(filedes, F_GETFL)) == -1)
    {
        printf("filestatus failed\n");
        return -1;
    }
    printf("File descriptor %d", filedes);
    switch(arg1 & O_ACCMODE)
    {
        case O_WRONLY:
           printf("write only");
           break;
        /* continua ... */
```

Esempio 1 d'uso di `fcntl` (II)

```
        case O_RDWR:
           print("read write");
           break;
        case O_RDONLY:
           print("read only");
           break;
        default:
           print("No such mode");
           break;
     }
     if(arg1 & O_APPEND) printf(" - append flag set");

     printf("\n");
     return 0;
}
```

- dove **O_ACCMODE** è una maschera appositamente definita in **<fcntl.h>**.

Esempio 2 d'uso di `fcntl`

- La funzione **`set_fl`** mette a 1 i flag specificati nel parametro **`flags`**
- Per comportarsi correttamente, richiede prima i flag correnti, poi utilizza la maschera con OR, ed infine salva i nuovi flag:

```
/* flags defines which file status flags to turn on */
void set_fl(int fd, int flags)
{
   int val;
   if ( (val = fcntl(fd, F_GETFL, 0)) < 0)
       err_sys("fcntl F_GETFL error");
   val |= flags;   /* turn on flags */
   if (fcntl(fd, F_SETFL, val) < 0)
       err_sys("fcntl F_SETFL error");
}
```

Funzione `ioctl`

```
int ioctl(int fd, int request, …);
```

- La funzione **`ioctl`** raccoglie tutti i comportamenti che non possono essere racchiusi nell'interfaccia standard basata su file
- In generale è usata per manipolare i parametri dei device
- **Categorie di operazioni**
  - disk labels I/O
  - file I/O
  - magnetic tapes I/O
  - socket I/O
  - terminal I/O

Esempio d'uso di `ioctl`

```
int main(int argc, char **argv)
{{
  int fd=STDIN_FILENO;
  struct winsize size;

  printf("Calling ioctl with TIOCGWINSZ ...\n");
  if (ioctl(fd,TIOCGWINSZ,(char *)&size) < 0) {
    perror("ERROR");

  printf("%d rows, %d columns\n",size.ws_row,size.ws_col);

  exit(EXIT_SUCCESS);
}
```

Funzione `fsync`

```
int fsync(int fd);
```

- La system call **`fsync`** effettua l'operazione di "flush" dei dati bufferizzati dal kernel per il file descriptor **`fd`**, ovvero li scrive sul disco o sul dispositivo sottostante
- Tale funzione esiste in quanto il gestore del file system può mantenere i dati nel buffer di memoria per diversi secondi (per ragioni efficienza), prima di scriverli su disco
- Ritorna 0 in caso di successo, -1 in caso di errore

Ulteriori system call per file e directory

- Oltre alle system call per leggere e scrivere file regolari (**`open, read, write, lseek, close`**), esistono system call per effettuare le seguenti operazioni:
  - leggere gli attributi di un file (**`stat`**)
  - modificare gli attributi di un file (**`chmod, chown, chgrp, …`**)
  - creare/eliminare hard link (**`link, unlink, remove`**)
  - creare e leggere le directory (**`mkdir, opendir, readdir, closedir`**)

System call `stat`

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

- Le tre funzioni ritornano un struttura **`stat`** contenente informazioni sul file
  - **`stat`** identifica il file tramite un filename
  - **`fstat`** identifica un file aperto tramite il suo descrittore
  - **`lstat`**, se applicato ad un link simbolico, ritorna informazioni sul link simbolico, non sul file linkato

Struttura `stat`

- **`stat`** è un puntatore ad una struttura di informazioni sul file specificato

```
struct stat {
    dev_t      st_dev;     // device
    ino_t      st_ino;     // inode
    mode_t     st_mode;    // protection
    nlink_t    st_nlink;   // number of hard links
    uit_t      st_uid;     // user ID of owner
    gid_t      st_gid;     // group ID of owner
    dev_t      st_rdev;    // device type (if node device)
    off_t      st_size;    // total size, in bytes
    blksize_t st_blksize;// best I/O block size
    blkcnt_t  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 status change
}
```

Il campo `st_mode`

- Nel campo **`st_mode`** c'è la codifica sul tipo di file
- Esistono delle macro per agevolare la determinazione del tipo di file a partire dal valore di **`st_mode`**:
  - è un file regolare? **`S_ISREG(m)`**
  - è una directory? **`S_ISDIR(m)`**
  - è un device a caratteri? **`S_ISCHR(m)`**
  - è un device a blocchi? **`S_ISBLK(m)`**
  - è una FIFO? **`S_ISFIFO(m)`**
  - è un link simbolico? **`S_ISLNK(m)`**
  - è un socket? **`S_ISSOCK(m)`**

Il campo `st_mode`

- ***set-user-ID* e *set-group-ID*:**
  - in **`st_mode`**, esiste un bit (*set-user-ID*) che fa in modo che quando questo file viene eseguito, l'*effective user id* prende il valore del campo **`st_uid`**
  - in **`st_mode`**, esiste un bit (*set-group-ID*) che fa in modo che quando questo file viene eseguito, l'*effective group id* prende il valore del campo **`st_gid`**
- **Questi due bit sono utilizzati per risolvere il problema di passwd:**
  - l'owner del comando **`passwd`** è **`root`**
  - quando **`passwd`** viene eseguito, il suo *effective user id* è uguale a **`root`**
  - il comando può modificare in scrittura il file **`/etc/passwd`**

Il campo `st_mode`

- **Costanti per accedere ai diritti di lettura e scrittura contenuti in `st_mode`**
  - `S_ISUID` set-user-ID
  - `S_ISGID` set-group-ID
  - `S_IRUSR` accesso in lettura, owner
  - `S_IWUSR` accesso in scrittura, owner
  - `S_IXUSR` accesso in esecuzione, owner
  - `S_IRGRP` accesso in lettura, gruppo
  - `S_IWGRP` accesso in scrittura, gruppo
  - `S_IXGRP` accesso in esecuzione, gruppo
  - `S_IROTH` accesso in lettura, altri
  - `S_IWOTH` accesso in scrittura, altri
  - `S_IXOTH` accesso in esecuzione, altri

I campi `st_size` e `st_block`

- Il campo **`st_size`** di **`stat`** contiene la dimensione effettiva del file
- Il campo **`st_block`** di **`stat`** contiene il numero di blocchi utilizzati per scrivere il file
  - dimensione “standard” di 512 byte
  - alcune implementazioni usano valori diversi (non portabile)
- il campo **`st_blksize`** di **`stat`** contiene la dimensione preferita per i buffer di lettura/scrittura

System call `truncate`

```
int truncate(char* pathname, off_t len);
int ftruncate(int filedes, off_t len);
```

- Queste due funzioni cambiano la lunghezza di un file, portandola alla dimensione specificata:
  - se la nuova dimensione è più corta, tronca il file alla dimensione specificata
  - se la nuova dimensione è più lunga, allunga il file alla dimensione specificata
- Non sono standard POSIX

I campi `st_atime`, `st_mtime` e `st_ctime`

- **Tre valori temporali sono mantenuti nella struttura `stat`**
  - **`st_atime`** (opzione **`-u`** in **`ls`**)
    - Ultimo tempo di accesso
    - Viene aggiornato a seguito di una **`read`**, **`creat/open`** (in caso di creazione di nuovo file), **`utime`** (trattata in seguito)
  - **`st_mtime`** (default in **`ls`**)
    - Ultimo tempo di modifica del contenuto
    - Viene aggiornato a seguito di una **`write`**, **`creat/open`**, **`truncate`**, **`utime`**
  - **`st_ctime`** (opzione **`-c`** in **`ls`**)
    - Ultimo cambiamento nello stato (nell'inode)
    - Viene aggiornato a seguito di una **`chmod`**, **`chown`**, **`creat`**, **`mkdir`**, **`open`**, **`remove`**, **`rename`**, **`truncate`**, **`link`**, **`unlink`**, **`utime`**, **`write`**
  - Attenzione ai cambiamenti su contenuto, inode e accesso per quanto riguarda le directory

System call `utime`

```
int utime(char* pathname, struct utimbuf *times);
struct utimbuf {
    time_t actime;
    time_t modtime;
}
```

- Questa system call modifica il tempo di accesso e di modifica di un file; il tempo di changed-status viene modificato automaticamente al nuovo valore

System call `access`

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

- Quando si accede ad un file, vengono utilizzati *effective uid* e *effective gid* per la verifica dei permessi di accesso
- In alcuni casi può essere necessario verificare l'accessibilità in base a *real uid* e *real gid*
- Per fare questo, si utilizza la system call **`access`**
- **`mode`** è un maschera ottenuta tramite bitwise OR delle seguenti costanti:
  - **`R_OK`** test per read permission
  - **`W_OK`** test per write permission
  - **`X_OK`** test per execute permission
  - **`F_OK`** test per esistenza file

System call `umask`

```
mode_t umask(mode_t cmask);
```

- Cambia la maschera di bit utilizzata per la creazione dei file; ritorna la maschera precedente; nessun caso di errore
- Per formare la maschera, si possono utilizzare le costanti **`S_IRUSR`**, **`S_IWUSR`**, ..., viste in precedenza
- Funzionamento:
  - La maschera viene utilizzata tutte le volte che un processo crea un nuovo file
  - Tutti i bit che sono accesi nella maschera, verranno spenti nell'access mode del file creato

System call `chmod`

```
int chmod (const char* path, mode_t mode);
int fchmod (int fildes, mode_t mode);
```

- Cambia i diritti di un file specificato dal **`path`** (**`chmod`**) o di un file già aperto (**`fchmod`**)
- Per cambiare i diritti di un file, l'effective uid del processo deve essere uguale all'owner del file oppure deve essere uguale a root

Ownership di un nuovo file

- Quando un file viene creato, l'access mode del file viene scelto in base al parametro **`mode`** delle system call **`open`** e **`creat`**
- L'owner del file viene posto uguale allo effective uid del processo
- POSIX permette due possibilità per il group id:
  - può essere uguale allo effective gid del processo
  - può essere uguale al group id della directory in cui il file viene creato
- In alcuni sistemi, questo può dipendere dal bit set-group-id della directory in cui viene creato il file

System call `chown`

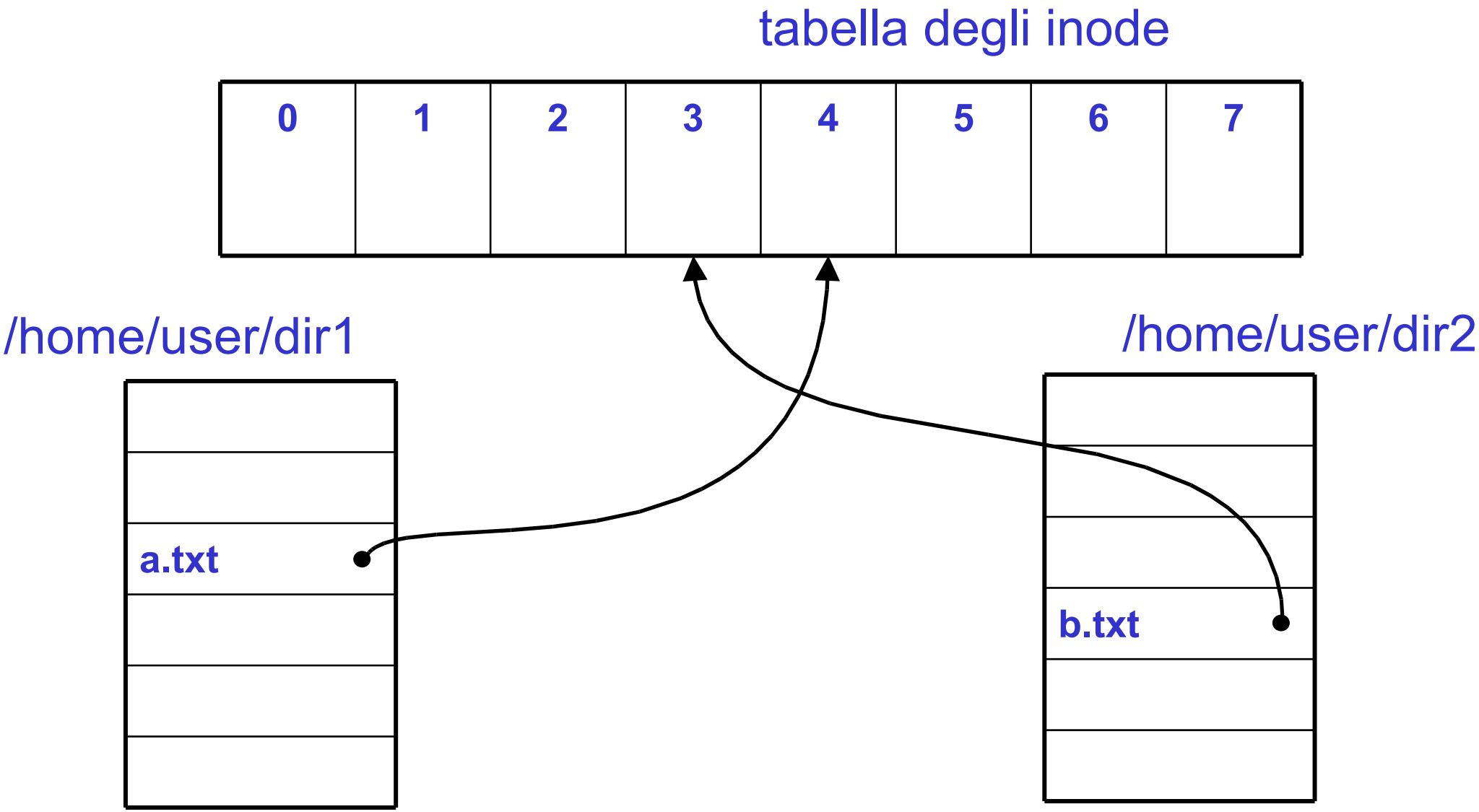
```
int chown(const char* pathname, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int lchown(const char* path, uid_t owner, gid_t group);
```

- **Queste tre funzioni cambiano lo user id e il group id di un file**
  - Nella prima, il file è specificato come **`pathname`**
  - Nella seconda, il file aperto è specificato dal file descriptor **`fd`**
  - Nella terza, si cambia il possessore del link simbolico, non del file stesso
- **Restrizioni:**
  - In alcuni sistemi, solo il superutente può cambiare l'owner di un file (per evitare problemi di quota)
  - Costante POSIX_CHOWN_RESTRICTED definisce se tale restrizione è in vigore

Esercizi 3

- A) Scrivere un programma che:
  - prende il nome di un file come argomento ed estrae le informazioni relative al tipo di file, bit di permesso, UID e GID, tempo dell'ultima modifica

Concetto di *hard link*



Concetto di *hard link*

- Nella figura precedente, sono mostrate due directory entry che puntano allo stesso i-node (due *hard link*)
- Ogni i-node mantiene un contatore con il numero di directory entry che puntano ad esso (numero di link)
- Operazioni:
  - Linking: aggiungere un link ad un file
  - Unlinking: rimuovere un link da un file
- Solo quando il numero di link scende a zero è possibile rimuovere il file (delete)
- Nella struttura **`stat`**: il campo **`st_nlink`** contiene il numero di link di un file

Concetto di *hard link*

- **Suddivisione delle informazioni:**
  - gli inode mantengono tutte le informazioni sul file: dimensioni, permessi, puntatori ai blocchi dati, etc.
  - le directory entry mantengono un file name e un inode number
- **Limitazioni degli hard-link:**
  - la directory entry contiene un numero di inode sullo stesso file system
  - non è possibile fare hard-linking con file su filesystem diversi
  - in questo caso, si utilizzano *symbolic link*

Hard link per directory

- Ogni directory **`dir`** ha un numero di hard link maggiore uguale a 2:
  - Un hard link esiste perché **`dir`** possiede un link nella sua directory genitore
  - Un'altro hard link esiste perché **`dir`** possiede un'entry "." che punta a se stessa
- Ogni sottodirectory di **`dir`** aggiunge un hard link:
    - Infatti la sottodirectory possiede un'entry ".." che punta alla directory genitore **`dir`**

System call `link`

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

- Crea un nuovo link ad un file esistente
- E' un'operazione atomica: aggiunge la directory entry e aumenta il numero di link per l'inode identificato da **`oldpath`**
- Cause di errore:
  - **`oldpath`** non esiste
  - **`newpath`** esiste già
  - solo root può creare un hard link ad una directory (per evitare di creare loop, che possono causare problemi)
  - **`oldpath`** e **`newpath`** appartengono a file system diversi
- E' utilizzata dal comando **`ln`**

System call `unlink` e `remove`

```
int unlink(char* path);
int remove(char* path);
```

- Rimuove un hard link per il file specificato da **`path`**
- E' un'operazione atomica: rimuove la directory entry e decrementa di uno il numero di link del file
- Cause di errore:
  - Il file **`path`** non esiste
  - un utente diverso da root cerca di fare **`unlink`** su una directory
- E' utilizzata dal comando **`rm`**
- Un file può essere effettivamente cancellato dall'hard disk solo quando il numero di hard link raggiunge 0

System call `unlink` e `remove`

- **Cosa succede quando un file è aperto?**
  - Il file viene rimosso dalla directory
  - Il file non viene rimosso dal file system fino alla sua chiusura
  - Quando il file viene chiuso, il sistema controlla se il numero di hard link è sceso a zero; in tal caso rimuove il file

System call `unlink` e `remove`

- **Come utilizzare questa proprietà:**
  - per assicurarsi che i file temporanei non vengano lasciati in giro in caso di crash del processo
  - sequenza di operazioni:
    - viene creato un file temporaneo
    - viene aperto dal processo
    - si effettua una operazione di unlink
  - Il file non viene cancellato fino al termine/crash del programma, che causa la chiusura di tutti i file temporanei

System call `rename`

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

- Cambia il nome di un file da **`oldpath`** a **`newpath`**
- Casi:
  - se **`oldpath`** specifica un file regolare, **`newpath`** non può essere una directory esistente
  - se **`oldpath`** specifica un file regolare e **`newpath`** è un file regolare esistente, questo viene rimosso e sostituito
  - sono necessarie write permission su entrambe le directory
  - se **`oldpath`** specifica una directory, **`newpath`** non può essere un file regolare esistente
  - se **`oldpath`** specifica una directory, **`newpath`** non può essere una directory non vuota

Link simbolici

- Un link simbolico è un file speciale che contiene il pathname assoluto di un altro file
- E' un puntatore indiretto ad un file (a differenza degli hard link che sono puntatori diretti agli inode)
- Introdotto per superare le limitazioni degli hard link:
  - hard link possibili solo fra file nello stesso filesystem
  - hard link a directory possibili solo al superuser
- **Nota:** questo comporta la possibilità di creare loop
  - i programmi che analizzano il file system devono essere in grado di gestire questi loop

Link simbolici

- Quando si utilizza una funzione, bisogna avere chiaro se:
  - segue i link simbolici (la funzione si applica al file puntato dal link simbolico)
  - non segue i link simbolici (la funzione si applica sul link)
- Funzioni che non seguono i link simbolici:
  - lchown, lstat, readlink, remove, rename, unlink
- Esempio (tramite shell):

```
$ ln -s /no/such/file myfile
$ ls myfile
myfile
$ ls -l myfile
lrwx------ penguin  myfile -> /no/such/file
$ cat myfile
cat: myfile: No such file or directory
```

System call `symlink` e `readlink`

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

- crea una nuova directory entry **`newpath`** con un link simbolico che punta al file specificato da **`oldpath`**
- **Nota**: **`ldpath`** e **`newpath`** non devono risiedere necessariamente nello stesso file system

```
int readlink(char* path, char* buf, int size);
```

- poiché la open segue il link simbolico, abbiamo bisogno di una system call per ottenere il contenuto del link simbolico
- questa system call copia in **`buf`** il valore del link simbolico

Operazioni su directory

- Le directory possono essere lette da chiunque abbia accesso in lettura
- Solo il kernel può scrivere in una directory per evitare problemi di consistenza
- I diritti di scrittura specificano la possibilità di utilizzare le system call: **`open, creat, unlink, remove, symlink, mkdir, rmdir`**
- Esistono delle funzioni speciali per leggere il contenuto di una directory: **`opendir, readdir`**

Directory corrente

- Ogni processo ha una directory corrente a partire dalla quale vengono fatte le ricerche per i pathname relativi

```
int chdir(char* path);

int fchdir(int filedes);
```

- Cambia la directory corrente associata al processo, utilizzando un pathname oppure un file descriptor

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

- Legge la directory corrente e riporta il pathname assoluto nel buffer specificato da **`buf`** e di dimensione **`size`**

Creazione e rimozione di directory

```
int mkdir(char* path, mode_t);
```

- Crea una nuova directory vuota dal path specificato
- Modalità di accesso:
  - I permessi specificati da mode_t vengono modificati dalla maschera specificata da umask
  - E' necessario specificare i diritti di esecuzione (search)

```
int rmdir(char* path);
```

- Rimuove la directory vuota specificata da path
- Se il link count della directory scende a zero, la directory viene effettivamente rimossa; altrimenti si rimuove la directory

Lettura di directory

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

- Apre una directory; ritorna un puntatore se tutto è ok, **NULL** in caso di errore

```
struct dirent *readdir(DIR *dir); // POSIX std function
```

- Ritorna un puntatore ad una struttura dati contenente informazioni sulla prossima entry, **NULL** se fine della directory o errore

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

- Ritorna all'inizio della directory

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

- Chiude la directory

Informazioni su directory

- La struttura **dirent** ritornata da **readdir** è definita in **dirent.h**:

```
struct dirent {
  ino_t d_ino;                // i-node number
  char  d_name[NAME_MAX+1];// null-terminated name
}
```

- **Note:**
  - la struttura **DIR** è una struttura interna utilizzata da queste quattro funzioni come "directory descriptor"
  - le entry di una directory vengono lette in un ordine non determinato a priori

Esercizi 4

- A) Scrivere un programma che:
  - discende ricorsivamente una gerarchia di directory e conta i vari tipi dei file.

Programma di gestione di un hotel (1/5)

- Sia **`residents.txt`** un file contenente la lista dei residenti di un hotel.
- La linea 1 di tale file contiene il nome della persona che occupa la camera 1, la linea 2 contiene il nome della persona che occupa la camera 2, etc.
- Ogni linea è lunga 41 caratteri, i primi 40 contengono il nome dell'occupante, l'ultimo è un newline (\n).
- Ecco un programma che stampa i nomi dei residenti in un albergo di 10 camere (pagg. seguenti).

Programma di gestione di un hotel (2/5)

```
#include <stdio.h>
#define NROOMS 10
main()
{
    int j;
    char *getoccupier(int), *p;
    for (j=1; j <= NROOMS; j++)
    {
        if (p = getoccupier(j))
           printf("Room %2d, %s\n", j, p);
        else
           printf("Error on room %d\n", j);
    }
}
```

Programma di gestione di un hotel (3/5)

```
/* getoccupier - restituisce il nome dell'occupante la camera
   passata come parametro */
#include <sys/types.h>
#include <unistd.h>
#define NAMELENGTH 41
char namebuf[NAMELENGTH]; /* buffer per contenere il nome */
int infile = -1; /* conterra` il file descriptor di
   residents.txt; l'inizializzazione serve affinche` venga
   aperto una sola volta */
/* continua ... */
```

Programma di gestione di un hotel (4/5)

```
char *getoccupier(int roomno)
{
    off_t offset; ssize_t nread;
    if (infile == -1 &&
        (infile = open("residents.txt", O_RDONLY)) == -1)
        return (NULL);
    offset = (roomno - 1) * NAMELENGTH;
    /* cerca la linea relativa alla camera e legge il nome */
    if (lseek(infile, offset, SEEK_SET) == -1)
        return (NULL);
    if ( (nread = read(infile, namebuf, NAMELENGTH)) <= 0)
        return (NULL);
    /* crea una stringa rimpiazzando il \n con \0 */
    namebuf[nread - 1] = '\0';
    return (namebuf);
}
```

Programma di gestione di un hotel (5/5)

- Come estensione del programma di gestione di un hotel, implementare un meccanismo per decidere se una camera è vuota, modificando eventualmente la funzione **`getoccupier`** e il file **`residents.txt`**. Scrivere una procedura **`findfree`** per trovare la prima camera libera.
- Scrivere le procedure:
  - **`freeroom`** per cancellare un occupante da una camera;
  - **`addguest`** per assegnare una camera ad un ospite, controllando che questa sia libera.
- Utilizzando le funzioni **`getoccupier, freeroom, addguest`**, e **`findfree`**, scrivere un programma **`frontdesk`** per gestire il file **`residents.txt`**.