Unix Systems Calls – I/O

dr. Andrea E. Naimoli

Università degli Studi di Trento

Dipartimento di Ingegneria e Scienza dell'Informazione

via Sommarive 14

I - 38050 Trento - Povo, Italy

I file: interfaccia standard Unix

Il sistema Unix fornisce due tipologie di interfacce per la programmazione attraverso file:

- **Stream**. Fornisce strumenti come la formattazione dei dati, bufferizzazione, ecc…
  - ✓ FILE* objects
- **File descriptors**. Interfaccia di basso livello costituita dalle system call messe a disposizione dal kernel
  - ✓ INT objects

stdio.h

Stream

- Un file è descritto da un puntatore a una struttura di tipo FILE (definita in **stdio.h**)
- I dati possono essere letti e scritti in vari modi (un carattere alla volta, una linea alla volte, ecc.) ed essere interpretati

File descriptors

- Un file è descritto da un semplice intero (file descriptor) che punta alla rispettiva entry *nella file table*
- I dati possono essere letti e scritti soltanto un buffer alla volta di cui spetta al programmatore stabilire la dimensione

File descriptors: Low Level I/O

Un insieme di system call permette di effettuare le operazioni di input e output mantenendo un controllo maggiore su quanto sta accadendo a prezzo di un'interfaccia meno amichevole.

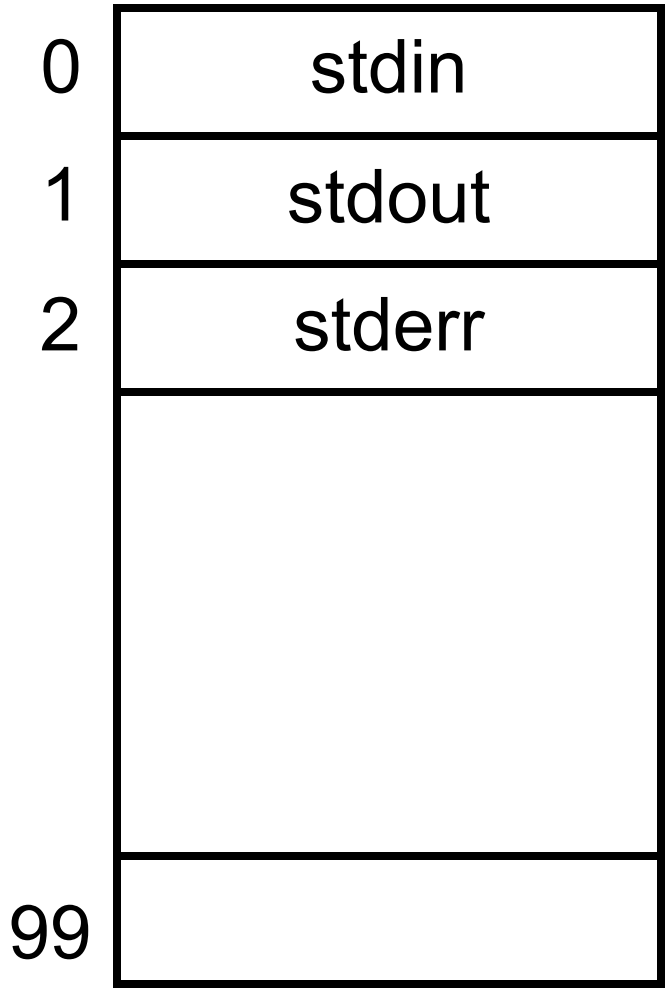
File descriptor

- Per accedere al contenuto di un file bisogna creare un canale di comunicazione con il kernel
- Questo si fa aprendo il file con la system call **open** la quale localizza l'i-node del file e aggiorna la file table del processo
- All'interno di ogni processo i file aperti sono descritti da un intero chiamato *file descriptor*
- In Unix ogni processo all'avvio ha tre file aperti, standard input (valore di *fd* 0), output (1), error (2)
- L'Input/Output Unix è basato essenzialmente su cinque funzioni: **open**, **read**, **write**, **lseek** e **close**

- Con la libreria stdio.h invece abbiamo stdin, stdout e stderr come file pointer predefiniti

Strutture dati del kernel (1/7)

- A ogni processo è associata **una tabella dei file aperti** di dimensione limitata (attorno al centinaio di elementi).
- Ogni elemento della tabella rappresenta un file aperto dal processo ed è individuato da un indice intero: file descriptor.
- I file descriptor 0,1,2 individuano rispettivamente standard input, output, error (aperti automaticamente).
- La tabella dei file aperti del processo è allocata nella sua user structure.



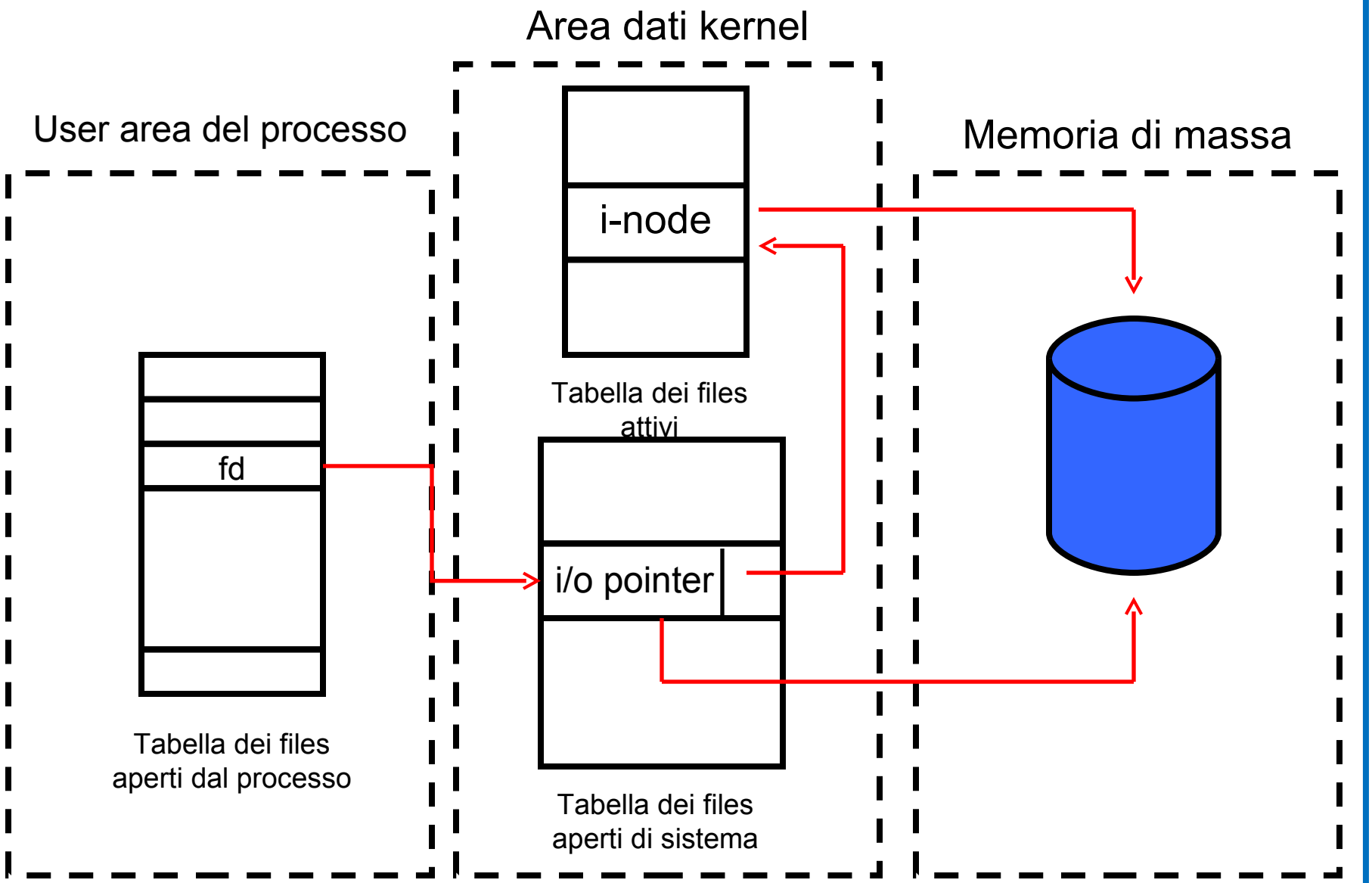
Strutture dati del kernel (2/7)

Per realizzare l'accesso ai file, SO utilizza due strutture dati globali, allocate nell'area dati del kernel.

- **La tabella dei file attivi**: per ogni file aperto, contiene una copia del suo i-node.
  - ✓ Più efficienti le operazioni su file evitando accessi al disco per ottenere attributi dei file acceduti (permessi, owner), informazioni temporali di modifica, etc…

- **La tabella dei file aperti di sistema**: ha un elemento per ogni operazione di apertura relativa a file aperti (e non ancora chiusi). Ogni elemento contiene
  - ✓ I/O pointer, posizione corrente all'interno del file
  - ✓ un puntatore all'i-node del file nella tabella dei file attivi
  - ✓ se due processi aprono separatamente lo stesso file **F**, la tabella conterrà due elementi distinti associati a F

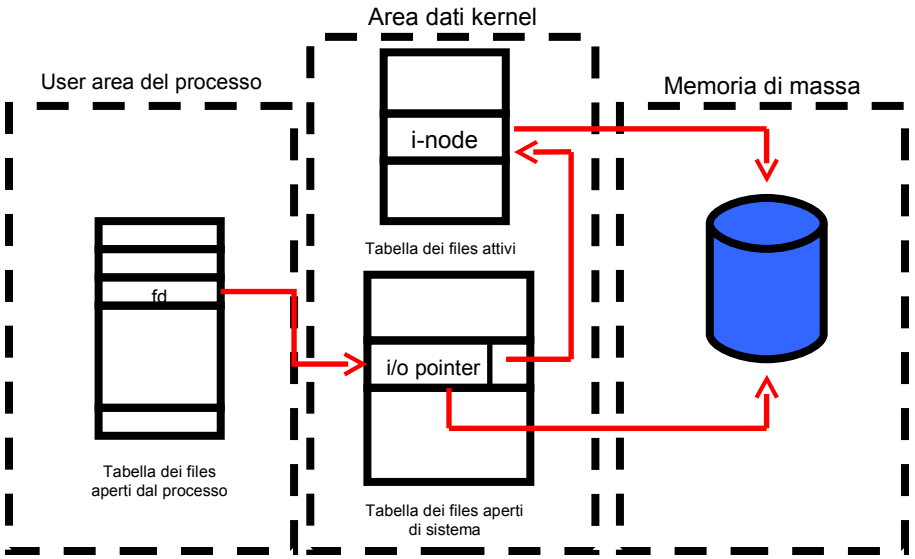
Strutture dati del kernel (3/7)



Strutture dati del kernel (4/7)

- **Tabella dei file aperti di sistema**

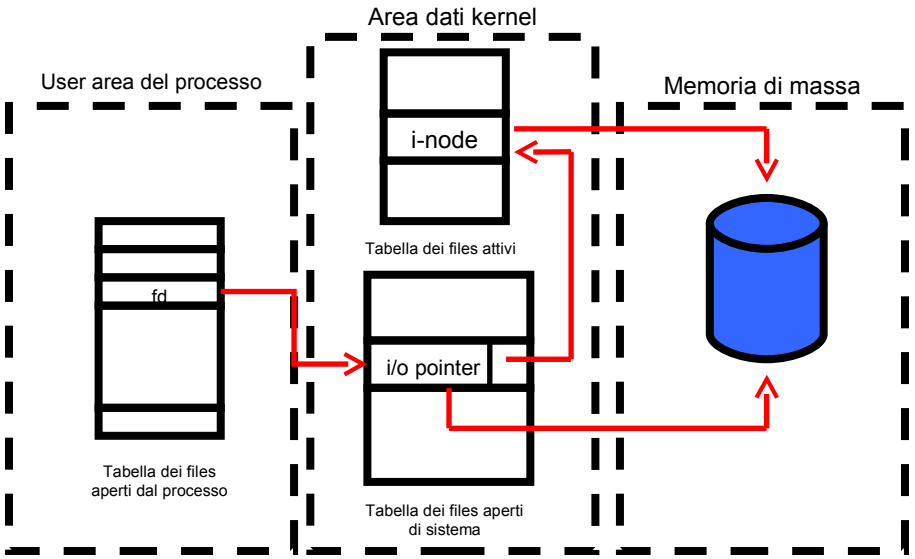
- Un elemento per ogni “apertura” di file
  - ✓ A processi diversi che accedono allo stesso file corrispondono entry distinte
- Ogni elemento contiene il puntatore alla posizione corrente (I/O pointer)
- Più processi possono accedere contemporaneamente allo stesso file, ma hanno I/O pointer distinti



Strutture dati del kernel (5/7)

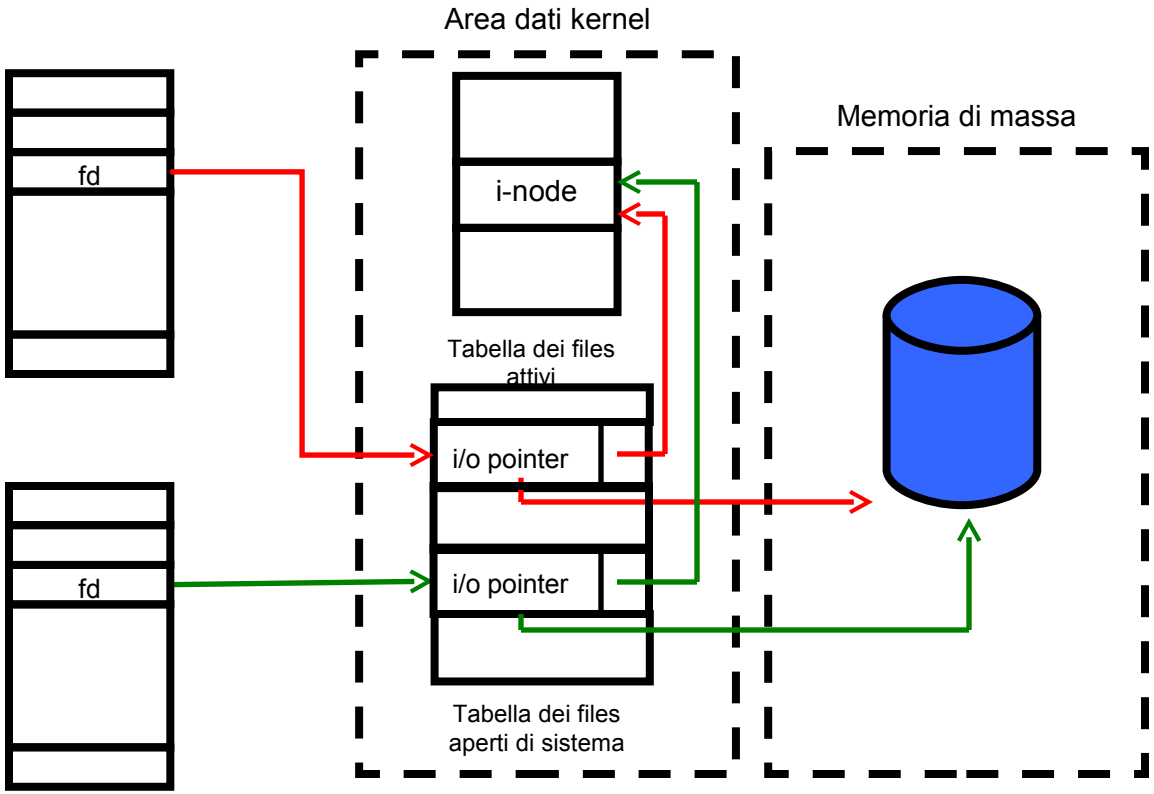
Tabella dei file attivi

- L'operazione di apertura provoca la copia dell'i-node dalla memoria centrale (se il file non è già in uso)
- La tabella dei file attivi contiene gli i-node di tutti i file aperti
- Il numero degli elementi è pari al numero dei file aperti (anche da più di un processo)



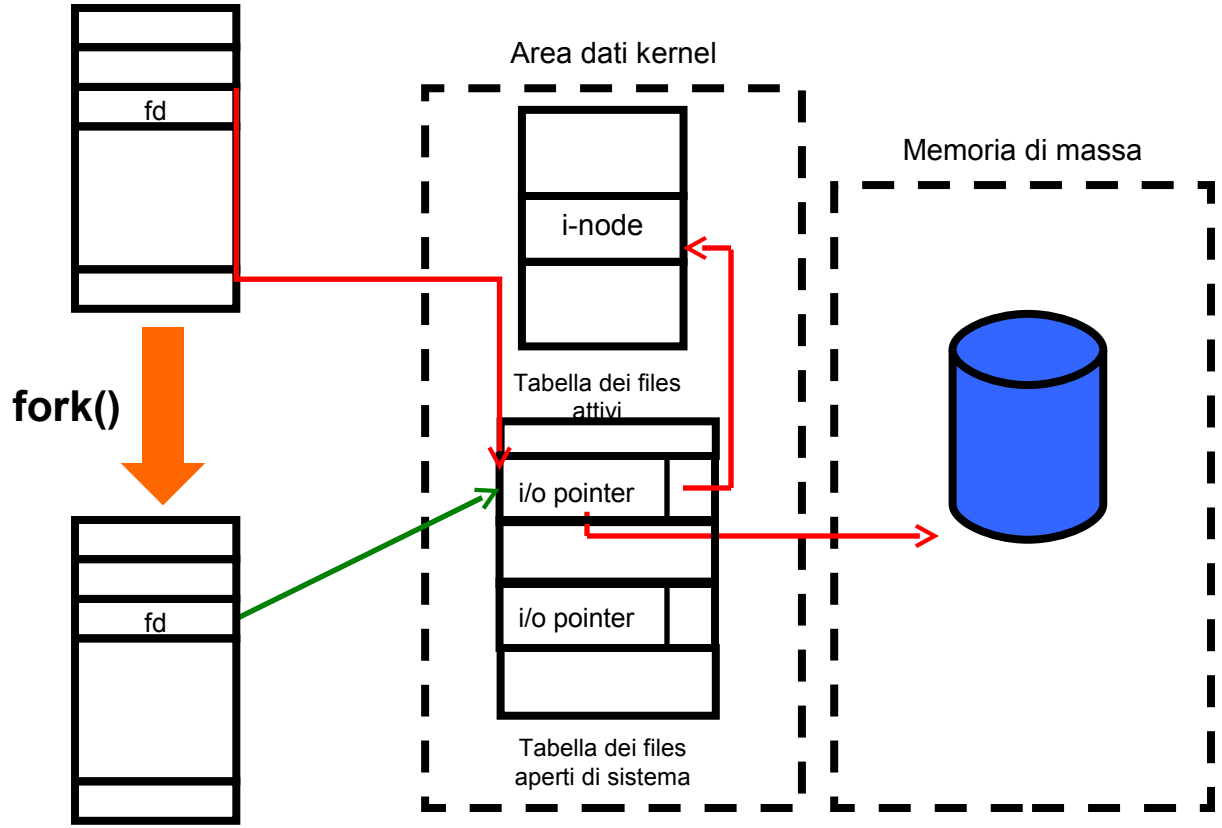
Strutture dati del kernel (6/7)

Esempio: processi A e B (indipendenti) accedono allo stesso file, ma con I/O pointer distinti



Strutture dati del kernel (7/7)

Esempio: processi A e B (indipendenti) accedono allo stesso file, ma con I/O pointer distinti



open (1/2)

```
int open(char *path, int flags, int mode)
int open(char *path, int flags)
```

- **path** path del file da aprire
- **flags** un intero che specifica in che modo aprire il file (#include <fcntl.h>)
- **mode** un intero che specifica le caratteristiche del file creato (#include <sys/stat.h>)

- L'intero ritornato è il file descriptor associato al file aperto (-1 in caso di errore)

Flag per open (2/2)

O_RDONLY	Read Only
O_WRONLY	Write Only
O_RDWR	Read + Write
O_APPEND	Scrive a partire dalla fine del file
O_CREAT	Crea il file se non esiste
O_EXCL	Usato con O_CREAT: errore se file esiste
O_TRUNC	Tronca il file se esiste

Con **O_APPEND** prima di ciascuna scrittura, la posizione corrente viene impostata dal kernel alla fine del file.

close

```
int close(int fd)
```

- **fd** file descriptor da chiudere

- 0 in caso di successo, -1 altrimenti
- Una close a buon fine NON garantisce che i dati siano effettivamente scritti su disco perché il kernel può deciderne il ritardo per ottimizzare l'accesso
- Per effettuare il flush dei dati si usa **sync** (da ripetere più di una volta)

lseek

off_t lseek(int fd, off_t offset, int whence)

- Ritorna l'offset in byte dall'inizio del file; -1 in caso di errore
- fd descriptor del file il cui offset va riposizionato
- offset valore di riposizionamento (relativo)
- whence specifica riferimento dell'offset
  - ✓ *SEEK_SET*: offset assoluto: si fa riferimento all'inizio del file
  - ✓ *SEEK_CUR*: offset calcolato dalla posizione corrente
  - ✓ *SEEK_END*: offset calcolato dalla fine del file (può essere negativo o positivo)

Esercizio (10 min.)

- **Scrivere un programma che visualizzi la dimensione di un file**
  - ✓ Il file deve essere specificabile dall'utente da linea di comando
  - ✓ Gestire ragionevolmente le possibili condizioni di errore (permessi, file non esistente, ecc.)

read

```
int read(int fd,void *buf,size_t nbytes)
```

- **fd** file descriptor da cui leggere
- **buf** puntatore a un buffer in cui mettere i dati letti dal file
- **nbytes** numero di byte da leggere dalla posizione corrente nel file

- L'intero ritornato è il numero di byte effettivamente letti dal file (0 se EOF, -1 in caso di errore)

write

int write(int fd, void *buf, int nbytes)

- **fd** file descriptor su cui scrivere
- **buf** puntatore a un buffer di dati da scrivere sul file
- **nbytes** numero di byte da scrivere

- L'intero ritornato è il numero di byte effettivamente scritti sul file (-1 in caso di errore)

Operazioni atomiche con i file

Il caso di scrittura su file in modalità “append”:

- Con la system call **lseek** è possibile impostare la posizione corrente all’offset desiderato (anche oltre la fine del file)
- La chiamata non causa nessun accesso al file, si limita a modificare la posizione corrente, ovvero il valore di **f_pos** nel file descriptor: alla successiva scrittura il file verrà semplicemente esteso
- in generale, se più processi distinti accedono contemporaneamente allo stesso file, le operazioni di lettura/scrittura sono locali al singolo processo ---> *race conditions*
- Il problema viene risolto aprendo il file in modalità **O_APPEND**: il kernel garantisce l’atomicità della sequenza **lseek**, **write** e quindi i dati vengono scritti alla fine del file

Altre system calls

```
int chown(const char *path, int owner,   int group);

int chmod(const char *path, int mode);

int chdir(const char *path);

int rmdir(const char *path);

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

Esercizio (30 min.)

- Scrivere un programma che funzioni come il comando UNIX *cp*
  - ✓ Aprire il file sorgente in lettura ed il file destinazione in scrittura
  - ✓ Leggere il file sorgente un buffer alla volta e scrivere quanto letto nel file destinazione
  - ✓ Chiudere i 2 file (attenzione alla gestione degli errori)

Esercizio per casa

- Scrivere un programma che prenda in input i nomi di 2 file e copi il primo in coda al secondo
- Se il secondo file non esiste deve essere creato con permessi -rw-r--r-- ed il primo file viene semplicemente copiato nel nuovo file
- Gli errori vanno gestiti in modo appropriato

Duplicare i file descriptor

- Il processo figlio derivante da una fork eredita i file descriptor aperti del genitore
- Quello che succede è che all'atto della fork la tabella dei file aperti dal processo viene duplicata
- In questo modo padre e figlio avranno lo stesso riferimento alla voce nella tabella condividendo così la posizione corrente nel file
- Questo conduce a una possibile race condition: due processi che scrivono sul medesimo file (un esempio sono i file di log)
- Il secondo scrive sul file fra una lseek e write del primo (non atomica)

dup

```
int dup(int fd);
```

- Prende come argomento il file descriptor di un file
- Restituisce un nuovo file descriptor per lo stesso file: sempre il più piccolo disponibile

dup2

```
int dup2(int fd_src, int fd_des);
```

- copia fd_src in fd_des
  - ✓ se fd_des esiste, viene chiuso
  - ✓ se fd_src = fd_des, i file descriptor non vengono chiusi
  - ✓ restituisce fd_des
- I 2 file descriptor condividono un solo file pointer

Come cambiare *stdin*

- ```
  close(0);
  fd = dup(0);
  ```

  che è analogo a (con fd impostato a un valore prescelto):

- ```
  dup2(0, fd);
  ```

- Lo stesso si può fare ovviamente per **stdout** (1) e **stderr** (2)

Esempio

- Voglio scrivere programma che faccia un ls e rediriga l'output sia su file sia su video
- Serve un processo che scrive su file e uno che scrive su video

Bibliografia

GaPiL. Guida alla Programmazione in Linux. Simone Piccardi

- *Modern Operating Systems*. Andrew. S. Tanenbaum, Prentice-Hall 1992, Edizione italiana *Moderni sistemi operativi*, Jackson Libri.
- *Linguaggio C*. Brian W. Kernighan and Dennis M. Ritchie
- *Sistemi Operativi – concetti e esempi*. Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. Sesta edizione, Addison-Wesley