

Input-Output di basso livello

I/O di basso livello

- La maggior parte delle operazioni sui file ordinari in ambiente UNIX si possono eseguire utilizzando solo le cinque chiamate di sistema **open, read, write, lseek, close**.
- Il kernel associa un *file descriptor* ad ogni file aperto
 - Il file descriptor è un numero intero
 - Quando un file viene aperto con `open`, questa funzione restituisce il file descriptor associato al file
- Le costanti simboliche `STDIN_FILENO` (0), `STDOUT_FILENO` (1) e `STDERR_FILENO` (2) sono definite in `unistd.h`

open

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *filename, int flags, ... /* mode_t mode */ )
```

- Restituisce il descrittore del file, o -1 in caso di errore
- Permette sia di aprire un file già esistente che di creare il file nel caso in cui questo non esista
- **filename** è il nome (assoluto o relativo) del file
- **flags** permette di specificare le opzioni, mediante costanti definite in <fcntl.h>, combinate con “|” (*or bit-a-bit*)
- **mode** specifica i permessi del file, nel caso che il file non esista e quindi debba essere creato da open

I flag di open

O_RDONLY	Aprire il file in sola lettura. Questa opzione non può essere combinata con O_WRONLY e O_RDWR.
O_WRONLY	Aprire il file in sola scrittura. Questa opzione non può essere combinata con O_RDONLY e O_RDWR.
O_RDWR	Aprire il file in lettura e scrittura. Questa opzione non può essere combinata con O_WRONLY e O_RDONLY.
O_APPEND	L'offset del file è posto alla fine del file prima di ogni chiamata a write
O_CREAT	Crea il file se esso non esiste. Richiede che open sia utilizzata con l'argomento mode, per la specifica del modo del file.
O_EXCL	Genera un errore se è stata specificata anche l'opzione O_CREAT e se il file già esiste.
O_TRUNC	Tronca il file a zero byte se esso esiste ed è stato aperto in sola scrittura o in lettura-scrittura.
O_NOCTTY	Se pathname si riferisce ad un terminale, non lo alloca come terminale di controllo per il processo.
O_NONBLOCK	Se pathname si riferisce ad una FIFO o ad un file di dispositivo, definisce la modalità nonblocking per l'apertura e l'I/O sul file.
O_SYNC	Specifica che ogni chiamata write deve attendere per il completamento dell'I/O sul dispositivo fisico.

I permessi per open

S_IRWXU	Permesso di lettura, scrittura ed esecuzione per il proprietario
S_IRUSR	Permesso di lettura per il proprietario
S_IWUSR	Permesso di scrittura per il proprietario
S_IXUSR	Permesso di esecuzione per il proprietario
S_IRWXG	Permesso di lettura, scrittura ed esecuzione per il gruppo
S_IRGRP	Permesso di lettura per il gruppo
S_IWGRP	Permesso di scrittura per il gruppo
S_IXGRP	Permesso di esecuzione per il gruppo
S_IRWXO	Permesso di lettura, scrittura ed esecuzione per gli altri
S_IROTH	Permesso di lettura per gli altri
S_IWOTH	Permesso di scrittura per gli altri
S_IXOTH	Permesso di esecuzione per gli altri

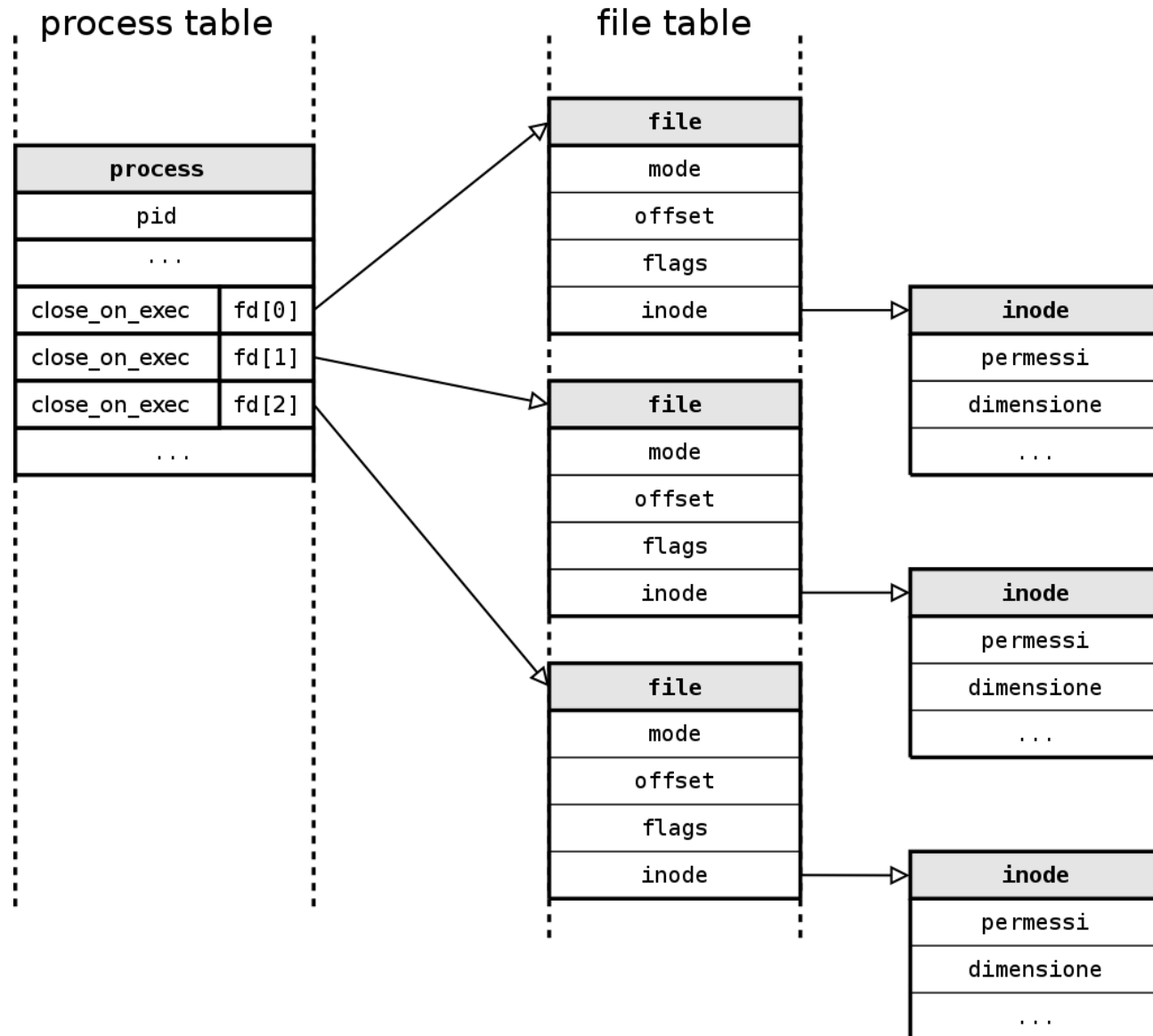
Esempi di open

- `open("prova.txt", O_RDONLY)`
- `open("prova.txt", O_RDONLY | O_CREAT, S_IRWXU)`
- `open("prova.txt", O_RDWR | O_CREAT | O_EXCL, S_IRWXU)`

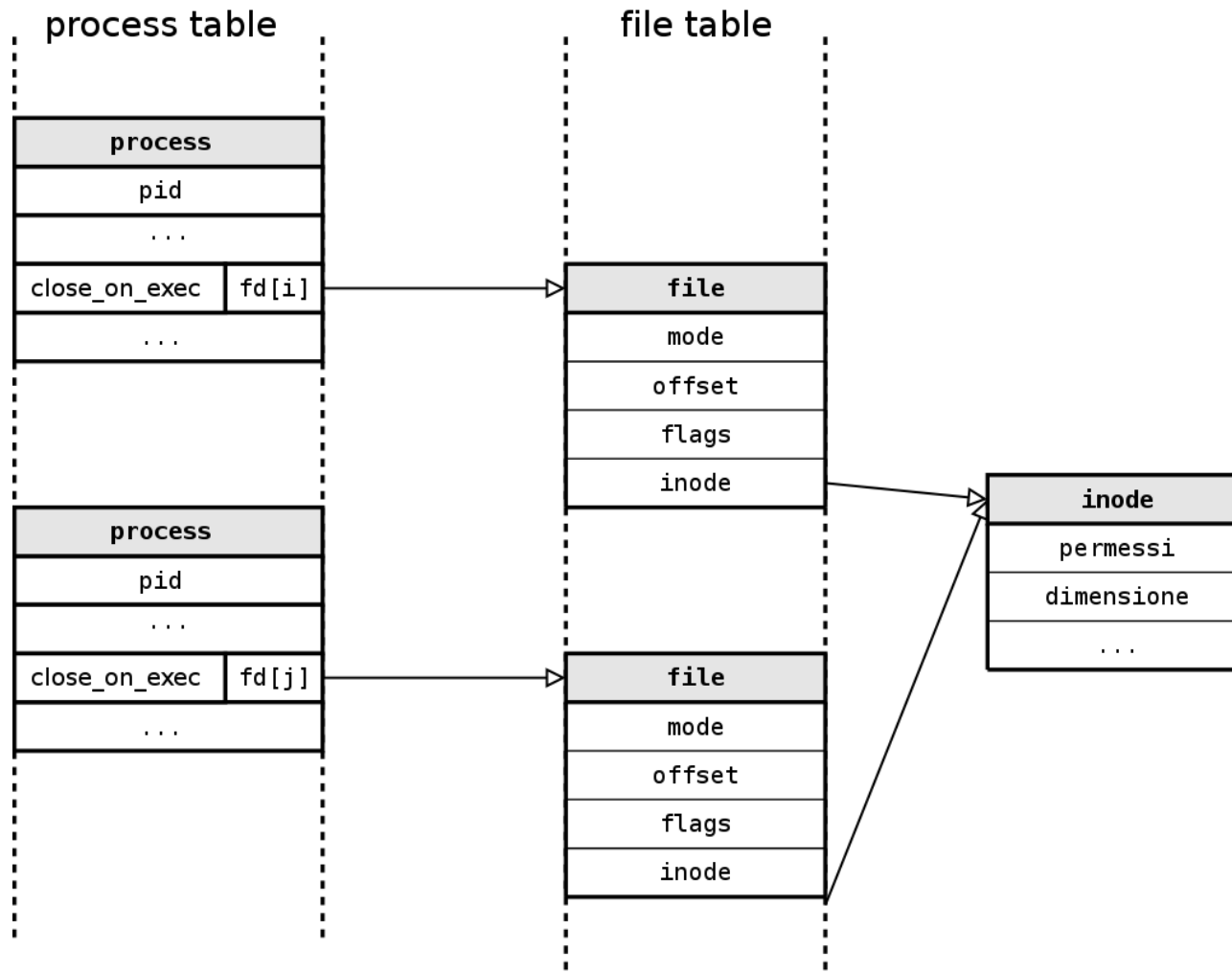
Implementazione nel kernel

- Il kernel usa due strutture dati indipendenti per gestire i file aperti
 - Ogni processo mantiene la **lista dei propri file descriptor**
 - Ogni file descriptor punta ad un elemento della **file table**
 - La file table specifica per ogni file aperto:
 - la modalità di apertura del file (lettura, scrittura o entrambe)
 - le opzioni come `O_APPEND`, etc.
 - l'offset corrente
 - l'inode corrispondente

Un processo con 3 descrittori aperti



Due processi che accedono allo stesso file



Trattare gli errori

- Molte system call restituiscono -1 in caso di errore
- Per avere più informazioni sull'errore, si usa la variabile globale ***errno*** (error number)
- La funzione ***perror(const char *)*** stampa la stringa passata come parametro, e poi un messaggio in base al valore corrente di ***errno***

- Esempi:

```
int fd = open("prova.txt", O_RDONLY);  
if (fd < 0) perror("errore di open");
```

```
int fd;  
if ( (fd=open("prova.txt", O_RDONLY)) < 0 )  
    perror("errore di open");
```

close

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

- Chiude il file identificato da filedes (e precedentemente aperto con open)
- Restituisce 0 in caso di successo e -1 in caso di errore

L'offset

- Ad ogni file aperto è associato un intero, detto *offset*, che rappresenta la posizione (espressa in numero di byte dall'inizio del file) in cui verrà effettuata la prossima operazione di I/O
- L'offset è inizializzato a **zero** da open
 - a meno che non sia specificato `O_APPEND`
- Le operazioni di read e write incrementano il valore dell'offset di un numero di byte pari al numero di byte letti/scritti

lseek

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

- Modifica l'offset corrente del file
- Restituisce il nuovo valore dell'offset, o -1 in caso di errore

lseek

Il valore del parametro **offset** è interpretato in base al parametro **whence**:

- SEEK_SET: L'offset corrente è posto a **offset** byte dall'**inizio** del file
- SEEK_CUR: L'offset corrente è incrementato di **offset** byte
 - Il valore del parametro **offset** può essere sia positivo che negativo
- SEEK_END: L'offset è posto a **offset** byte dalla **fine** del file
 - Il valore del parametro **offset** può essere sia positivo che negativo

Quindi, per conoscere l'offset corrente, è sufficiente eseguire:

```
x = lseek(filedes, (off_t)0, SEEK_CUR);
```

read

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

- Restituisce:
 - il numero di byte effettivamente letti
 - 0 se ci troviamo alla fine del file
 - -1 in caso di errore
- L'operazione di lettura avviene partendo dall'offset corrente del file
 - l'offset viene incrementato opportunamente

read

Il numero di byte letti può essere diverso dal parametro nbytes quando:

- Il numero di byte ancora presenti nel file è inferiore ad nbytes
- La lettura avviene da un terminale
- La lettura avviene da un buffer di rete
- La lettura avviene da una pipe o una FIFO
- L'operazione è interrotta da un segnale.

write

```
ssize_t write(int filedes, void *buff, size_t nbytes)
```

- Restituisce:
 - il numero di byte effettivamente scritti
 - -1 in caso di errore
- L'operazione di scrittura avviene partendo dall'offset corrente del file
 - l'offset viene incrementato opportunamente

Esempio 9

```
#include <stdio.h>      /* perror */
#include <errno.h>      /* perror */
#include <unistd.h>     /* write, lseek, close, exit */
#include <sys/types.h>  /* open, lseek */
#include <sys/stat.h>   /* open */
#include <fcntl.h>     /* open */
```

```
char    buf1[] = "abcdefghij";
char    buf2[] = "ABCDEFGHIJ";
```

```
int main(void)
{
    int fd;
```

Esempio 9

```
if ((fd = open("file.hole", O_RDWR|O_CREAT, S_IRWXU)) < 0)
    perror("open error");
if (write(fd, buf1, 10) != 10)
    perror("buf1 write error");
/* L'offset ora e' 10 */

if (lseek(fd, 20, SEEK_SET) == -1)
    perror("lseek error");
/* L'offset ora e' 20 */

if (write(fd, buf2, 10) != 10)
    perror("buf2 write error");
/* L'offset ora e' 30 */

close(fd);
return 0;
}
```

Esempio 10

```
#include <stdio.h>      /* perror */
#include <errno.h>      /* perror */
#include <unistd.h>     /* read, write */
#define BUFFSIZE 4096

int main(void) {
    int n;
    char buf[BUFFSIZE];

    while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            perror("write error");

    if (n < 0) perror("read error");

    return 0;
}
```

Esercizi

- Scrivere un programma che mostra il contenuto di un file a byte alterni (un carattere sì e uno no)
- Scrivere un programma che mostra il contenuto di un file alla rovescia, cioè a partire dall'ultimo carattere fino ad arrivare al primo