

Unbuffered I/O

Ing. Giovanni Nardini - University of Pisa - All rights reserved

Parliamo delle system call per fare operazioni di input e output. Si chiamano “unbuffered” perché il programma effettua operazioni direttamente sui file/dispositivi (lo può fare perché parliamo, appunto, di system call) senza “intermediari”, come ad esempio dei buffer nel kernel.

File I/O

Operazioni sui file → apertura, lettura, scrittura, ecc.

- In Unix *tutto è un file*

- *Regular files*
- *Input/output streams*
- *Symbolic links*
- *Devices (e.g. disk)*
- *Named pipes*
- *Sockets*
- ...



Saper lavorare sui file ci permette di operare con tutto il sistema

- I file in Unix non hanno una struttura interna

- La loro interpretazione è demandata al processo che li usa



Lavorare sui file è semplice

Si può fare (quasi) tutto con solo **5** system call!

Ci riferiamo a operazioni di I/O su file. Questo non è una limitazione, perché sappiamo che in Unix tutto può essere rappresentato come un file, compresi i dispositivi di input/output (come tastiera e schermo).

Un file in Unix è semplicemente una sequenza di byte, per cui le system call che vedremo si occupano di leggere/scrivere byte senza preoccuparsi di rispettare una qualche struttura/formattazione.

File descriptor

- Dal punto di vista di un processo, ciascun file è identificato da un numero → **file descriptor**
 - Numero intero
 - Non negativo
 - Piccolo
- All'interno di un programma, si utilizza il file descriptor per interagire col file
- Gli stream standard di input e output hanno un **file descriptor predefinito**
 - 0 (input), 1 (output), 2 (error)
 - Consigliato l'uso di costanti
 - `stdin, stdout, stderr` definite dalla libreria standard C in `<stdio.h>`
 - `STDIN_FILENO, STDOUT_FILENO, STDERR_FILENO` definite dallo standard POSIX in `<unistd.h>`

Ogni volta che il programma deve fare un'operazione su un file, deve conoscere o ottenere un file descriptor per potersi interfacciare con esso.

Ci ricordiamo che ogni processo possiede tre stream predefiniti: `stdin`, `stdout` e `stderr`. Così come gli altri oggetti possono essere astratti come file, per cui possiamo interagire con tali stream utilizzando i loro file descriptor. Un programma che vuole leggere da `stdin` o scrivere su `stdout`, utilizza le funzioni di sistema per la lettura/scrittura su file, riferendosi ai file descriptor di `stdin/stdout`

System call per operazioni di I/O

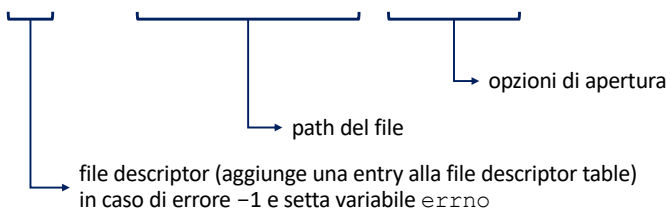
- Usa solamente cinque **system call**
 - open
 - close
 - read
 - write
 - lseek
- Presenti nel manuale Unix: `man 2 open`, `man 2 read`, ...
- Definite dallo standard POSIX

Queste system calls sono definite in diversi header dello standard POSIX

Apertura di un file

```
#include <fcntl.h>
```

```
int open(const char* path, int oflags);
```



| | |
|-----------------------|-----------------------------|
| <code>O_RDONLY</code> | Apri in lettura |
| <code>O_WRONLY</code> | Apri in scrittura |
| <code>O_RDWR</code> | Apri in lettura e scrittura |

```
int fd = open("dummy", O_RDONLY);
```

```
int fd = open("dummy", O_WRONLY);
```

```
int fd = open("dummy", O_WRONLY | O_APPEND);
```

Possono essere messe in OR con:

| | |
|-----------------------|---------------------------------------|
| <code>O_APPEND</code> | Append alla fine del file |
| <code>O_CREAT</code> | Crea il file, se non esiste |
| <code>O_TRUNC</code> | Tronca la lunghezza del file a 0 byte |
| ... | ... |

Aprire il file significa **ottenere un file descriptor** associato al file, per poter fare poi delle operazioni di I/O su di esso.

Dobbiamo specificare quale tipo di operazioni voglio fare su quel file descriptor (lettura, scrittura o entrambe) tramite l'argomento *oflags*.

Il tipo di operazioni (detto anche modalità di accesso) può essere messo in OR con ulteriori opzioni (append, crea se il file non esiste, ...)

Apertura di un file

Corretta gestione degli errori:


```
#include <fcntl.h>           // for O_RDONLY
#include <stdio.h>           // for fprintf and stderr
#include <errno.h>           // for errno
#include <string.h>          // for strerror
#include <stdlib.h>          // for exit and EXIT_FAILURE

...

int fd = open("dummy", O_RDONLY);
if (fd == -1) {
    fprintf(stderr, "Error while opening file: %s", strerror(errno));
    exit(EXIT_FAILURE);
}

...
```

File non esiste, utente non ha i permessi,...



Unbuffered I/O

Ing. Giovanni Nardini - University of Pisa - All rights reserved

37

Dopo ogni chiamata alla system call devo controllare il risultato per verificare che non ci siano errori.

Non posso leggere da un file se il file descriptor non è stato ottenuto correttamente (per esempio se il file specificato non esiste)

Creazione di un file

```
#include <fcntl.h>
```

```
int open (const char* path, int oflags, mode_t mode);
```

- Quando si usa il flag `O_CREAT`, è necessario specificare i permessi sul file
- Se si specifica anche il flag `O_EXCL`, genera un errore se il file esiste già

permessi

| | |
|---------|-------------------------|
| S_IRUSR | Lettura proprietario |
| S_IWUSR | Scrittura proprietario |
| S_IXUSR | Esecuzione proprietario |
| S_IRGRP | Lettura gruppo |
| S_IWGRP | Scrittura gruppo |
| S_IXGRP | Esecuzione gruppo |
| S_IROTH | Lettura altri utenti |
| S_IWOTH | Scrittura altri utenti |
| S_IXOTH | Esecuzione altri utenti |

```
int fd = open("dummy", O_RDONLY|O_CREAT, S_IRUSR|S_IWUSR);
```

```
int fd = open("dummy", O_WRONLY|O_CREAT|O_EXCL, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
```

L'argomento `mode` può assumere una o più costanti messe in OR tra loro.
Notare i tre gruppi di permessi: utente (USR) / Gruppo (GRP) / altri (OTH)

Se uso il flag `O_CREAT` con un file che esiste già non ho errori, a meno che non abbia inserito anche il flag `O_EXCL`. Perché mettere `O_EXCL` allora?

Chiusura di un file

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

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

0 se ok, altrimenti -1 e setta variabile `errno`

file descriptor

```
int fd = open("dummy", O_RDONLY);
```

```
...
```

```
close(fd);
```

- Tutti i file descriptor vengono comunque rilasciati implicitamente al termine del processo

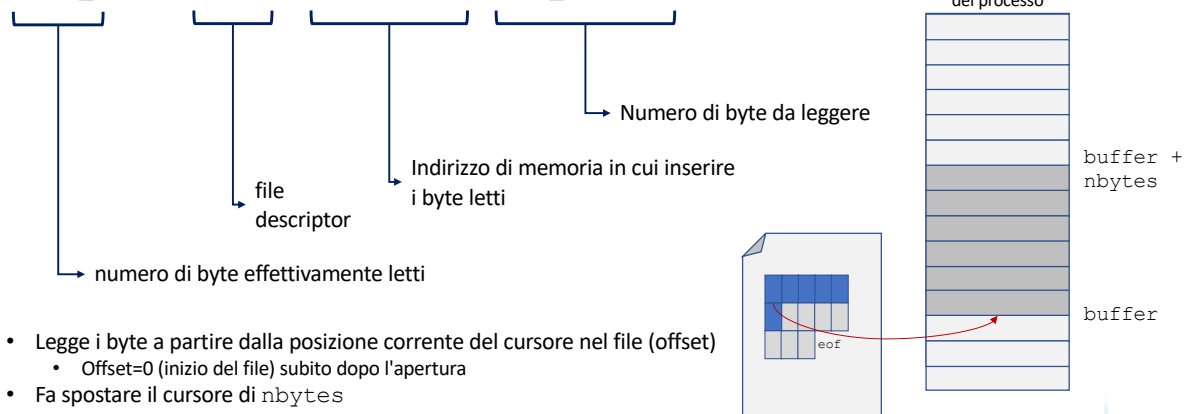
Quando un file non serve più, ovvero abbiamo fatto su di esso tutte le operazioni che ci servivano, possiamo chiuderlo, cioè *liberare* il file descriptor

Perché è necessario? Esiste un numero massimo di file descriptor che possono essere aperti contemporaneamente da un processo. Inoltre, quando un file descriptor è aperto, si creano delle strutture dati in memoria: tenerle aperte all'infinito occupa spazio di memoria inutilmente.

Lettura da file

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

```
ssize_t read(int fd, void* buffer, size_t nbytes);
```



- Legge i byte a partire dalla posizione corrente del cursore nel file (offset)
 - Offset=0 (inizio del file) subito dopo l'apertura
- Fa spostare il cursore di `nbytes`

Unbuffered I/O

Ing. Giovanni Nardini - University of Pisa - All rights reserved

41

A ogni file «aperto» viene associato un **offset**, che rappresenta la posizione corrente nel file (numero di byte dall'inizio del file).

L'offset rappresenta il prossimo byte che sarà letto dalla funzione `read`.

La system call `read()` legge **nbytes** dal file (a partire dall'offset) e li inserisce nella memoria del processo, all'indirizzo specificato da **buffer**. L'offset viene aggiornato di conseguenza, così che la prossima lettura avvenga da quel punto del file in avanti.

La funzione può leggere anche meno di `nbytes` (ad esempio nel caso che venga raggiunta la fine del file, rappresentata dal carattere speciale **EOF** (end-of-file)).

Possibili errori:

- il file descriptor è stato aperto in sola scrittura
- il file descriptor è già stato chiuso

Lettura da file

```
#include <unistd.h>
...
const int BUFFSIZE = 4096;
char buff[BUFFSIZE];
int n = read(fd, buff, BUFFSIZE);
if (n == -1) {
    ...    // errore
}
printf("Read %d bytes\n", n);
```

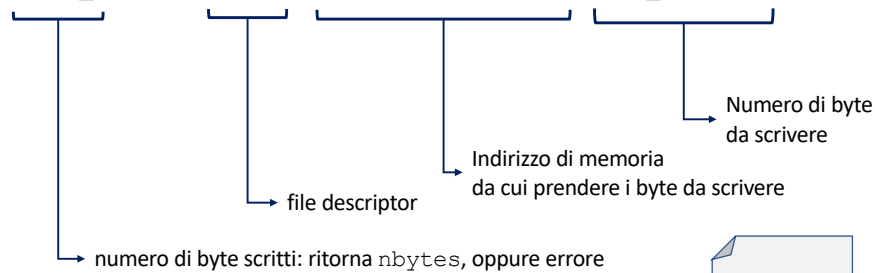
Operazione preliminare prima di effettuare la read: preparare un buffer, ovvero un'area di memoria del processo dove verranno inseriti i byte letti.

Siccome un char occupa 1 byte, la norma è dichiarare un array di caratteri della dimensione desiderata.

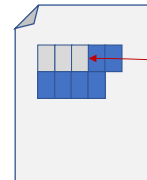
Scrittura su file

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

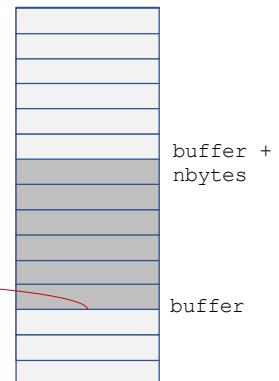
```
ssize_t write(int fd, const void* buffer, size_t nbytes);
```



- Scrive i nuovi byte alla posizione corrente del cursore nel file (offset)
 - Generalmente all'inizio del file dopo l'apertura
 - Se aperto con `O_APPEND`, l'offset è la fine del file
- Fa spostare il cursore di `nbytes`



Rappresentazione della memoria



Unbuffered I/O

Ing. Giovanni Nardini - University of Pisa - All rights reserved

43

In questo caso, l'offset rappresenta la posizione in cui verrà scritto il prossimo byte

La system call `write()` scrive nel file (a partire dall'offset) **nbytes** presenti in memoria del processo a partire dall'indirizzo specificato da **buffer**. L'offset viene aggiornato di conseguenza, così che la prossima scrittura avvenga da quel punto del file in avanti.

Differenza con `read()`: `write()` scrive tutti gli `nbytes` oppure nessuno (non può scrivere solo una parte degli `nbytes`), per cui restituisce `nbytes` oppure `-1` (errore).

Possibili errori:

- il file descriptor è stato aperto in sola lettura
- il file descriptor è già stato chiuso

Scrittura su file

```
#include <unistd.h>
#include <string.h> // for strlen

...
const char* message = "Trust me, I'm an engineer\n";
int len = strlen(message);
int n = write(fd, message, len);
if (n == -1) {
    ... // errore
}
printf("Written %d bytes\n", n);
...
```

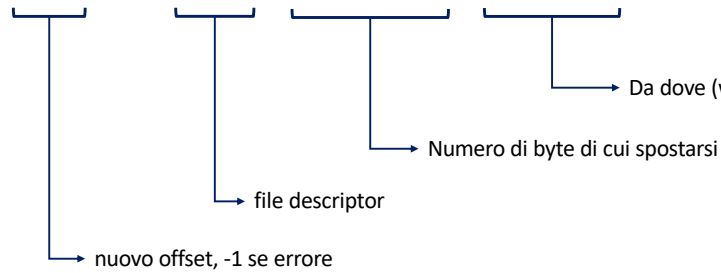
Operazione preliminare prima di effettuare la `write()`: preparare un buffer dentro cui è presente ciò che vogliamo scrivere sul file.

Tipicamente, si usa una stringa - e la sua dimensione si ottiene con `strlen()`

Modificare l'offset di un file

```
#include <fcntl.h>
```

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



| | |
|----------|--------------------------|
| SEEK_SET | Dall'inizio del file |
| SEEK_CUR | Dalla posizione corrente |
| SEEK_END | Dalla fine del file |

```
int new_offset = lseek(fd, 10, SEEK_CUR);
```

`lseek()` permette di modificare la posizione corrente nel di un certo numero di bytes, specificato dall'argomento **offset**, senza effettuare operazioni di lettura/scrittura. L'argomento `offset` è relativo rispetto all'argomento **whence**.

Possibile errore:

- proviamo a spostarsi a una posizione negativa (prima dell'inizio del file)
- file descriptor già chiuso
- file descriptor non seek-abile (vedi slide successiva)

Modificare l'offset di un file

Casi particolari:

```
int pos = lseek(fd, -10, SEEK_END);
```

Offset negativo

```
int pos = lseek(fd, 0, SEEK_CUR);
```

Restituisce la posizione corrente

```
int pos = lseek(fd, 1000, SEEK_SET);
```

Se oltrepassa la fine del file, la prossima scrittura crea un "buco"

- Non tutti i tipi di file sono "seek"-abili
 - Socket, pipes, ...

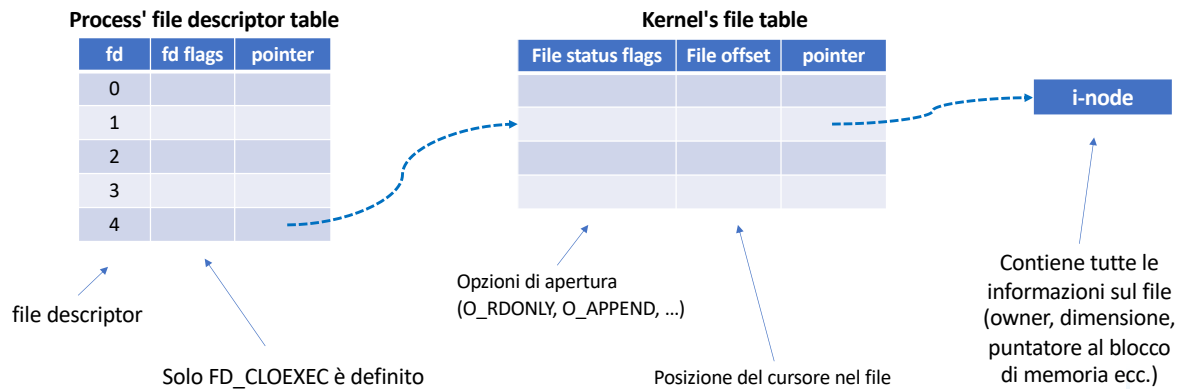
Il primo esempio fallisce se il file è più corto 10 byte

Il secondo esempio non sposta l'offset, a cosa serve?

Alcuni tipi di file non sono seek-abili: non è possibile effettuare la chiamata `lseek()` su un file descriptor che li rappresenta, perché tali oggetti hanno a che fare con stream di dati (ad esempio un file di tipo socket ricevere un flusso di dati dalla rete, non è possibile "seek"are qualcosa che non è ancora arrivato)

File descriptor table

- Ogni processo possiede una propria **file descriptor table**
- Il kernel mantiene una **file table** globale per il sistema



Unbuffered I/O

Ing. Giovanni Nardini - University of Pisa - All rights reserved

47

La kernel file table è unica per tutto il sistema, mentre ogni processo ha una propria file descriptor table.

Entrambi i tipi di tabella risiedono nel kernel space del sistema. Perché?

All'avvio del processo, la tabella dei file descriptor contiene – di default – i tre file descriptor 0,1 e 2 per stdin, stdout, stderr.

L'i-node è una struttura dati memorizzata nel file system del sistema che contiene i metadati del file (l'i-node NON è il file)

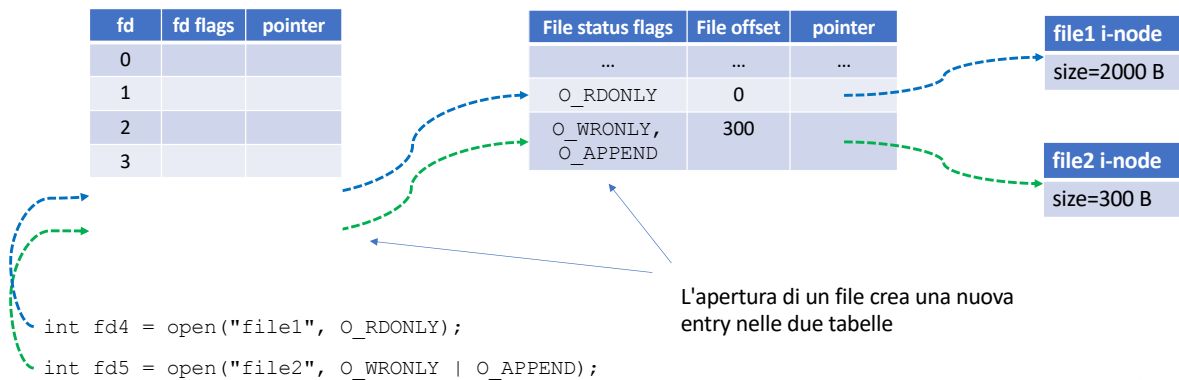
i-node

- Nel file system di Unix, ogni file è identificato dal suo **i-node**
- Descrive gli attributi del file:
 - tipo
 - utente proprietario
 - gruppo proprietario
 - dimensione
 - data di creazione/modifica/accesso
 - numero di hard link
 - permessi
 - puntatore/i al contenuto effettivo del file

- Grande assente: nome del file
- Dove le abbiamo viste queste informazioni?

File descriptor table

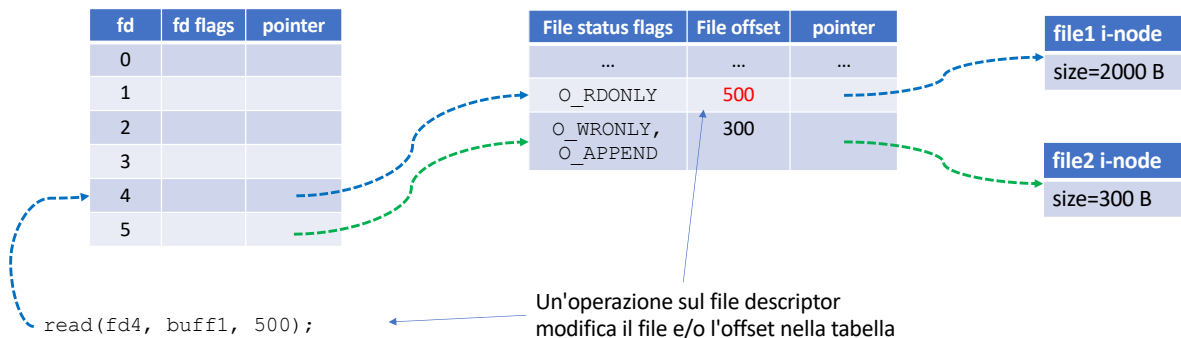
Apertura di un file



Quando si invoca la system call `open()`, viene creata una nuova riga nella tabella dei file descriptor – nella prima posizione libera - e una nuova riga nella kernel file table.

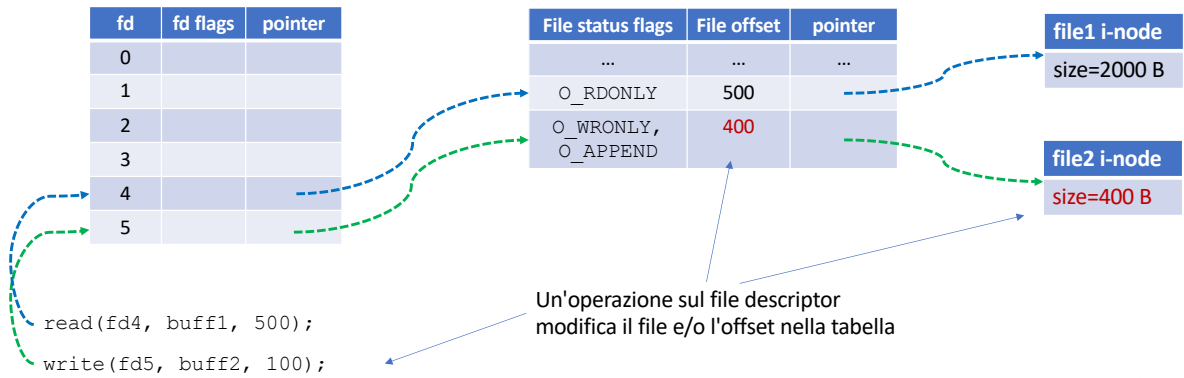
File descriptor table

Lettura da un file



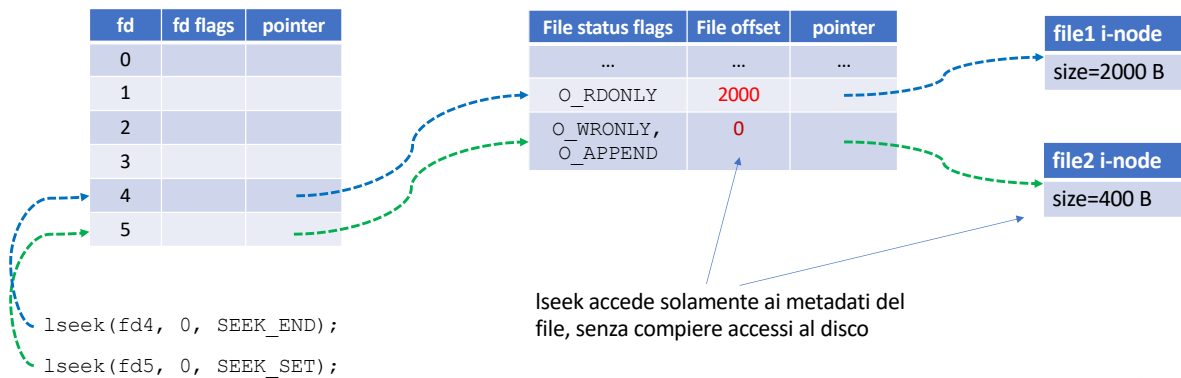
File descriptor table

Scrittura su un file



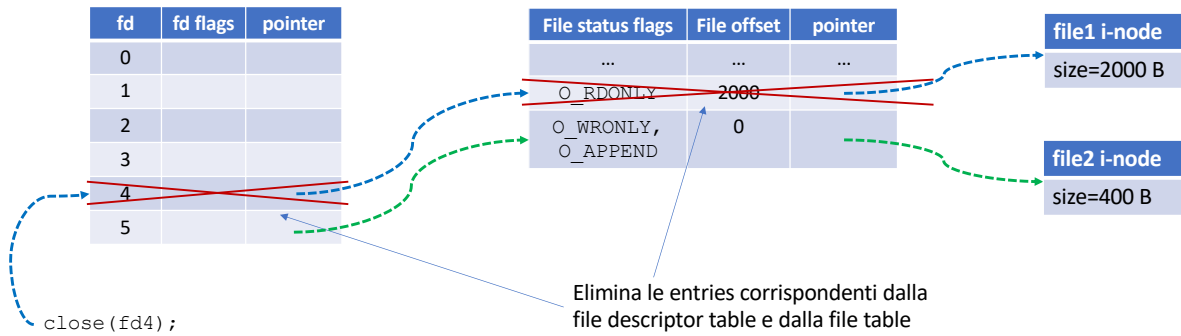
File descriptor table

Spostamento dell'offset



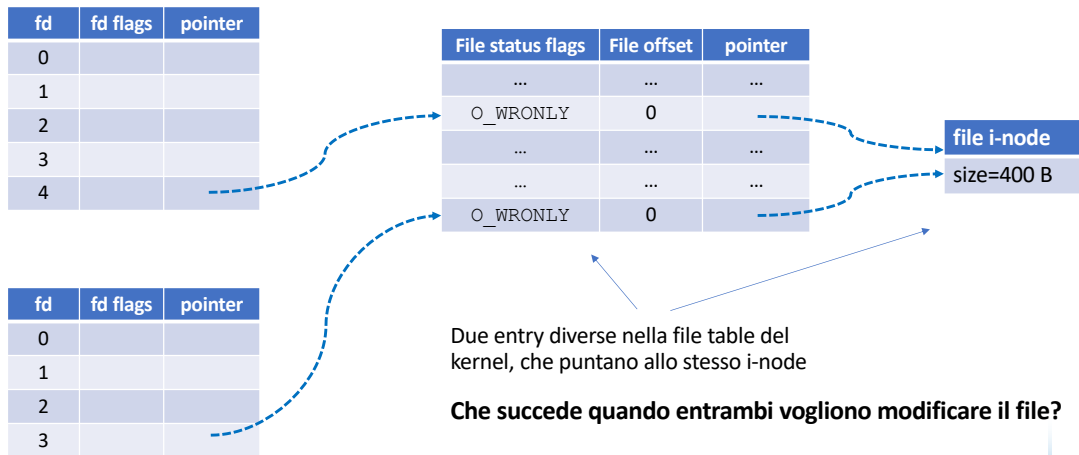
File descriptor table

Chiusura di un file



Condivisione di file tra processi

Due processi possono aprire lo stesso file contemporaneamente



Unbuffered I/O

Ing. Giovanni Nardini - University of Pisa - All rights reserved

54

I file descriptor nelle tabelle dei file descriptor hanno “visibilità” locale al processo stesso: si possono riutilizzare gli stessi fd per file diversi, uno stesso file può essere indicato da due fd diversi.

Condivisione di file tra processi

Esempio: entrambi i processi A e B si vogliono spostare alla fine del file e scrivere 100 byte

Proc. A:

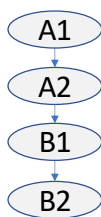
```
1: lseek(fd, 0, SEEK_END);  
2: write(fd, buf, 100);
```

Proc. B:

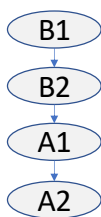
```
1: lseek(fd, 0, SEEK_END);  
2: write(fd, buf, 100);
```

In quale ordine vengono eseguite le istruzioni?

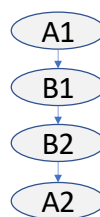
Non possiamo saperlo!



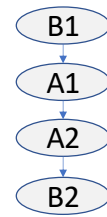
OK!



OK!



!!!



!!!

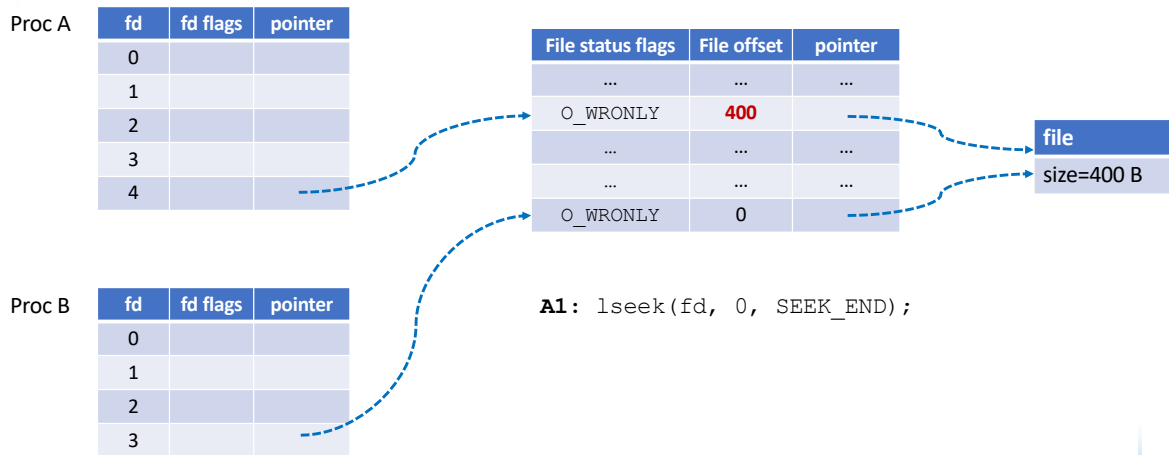
Supponiamo che i processi A e B eseguono un programma in cui è previsto lo spostamento alla fine del file (con una `lseek`) e la scrittura di 100 byte (con una `write`), e supponiamo che il sistema abbia un solo processore (per cui i due processi devono essere eseguiti concorrentemente).

Il sistema operativo (lo scheduler) può decidere di passare da A a B (o viceversa) in ogni momento: in figura sono rappresentati i 4 casi possibili di esecuzione.

Negli ultimi due casi potrebbero sorgere alcuni conflitti.

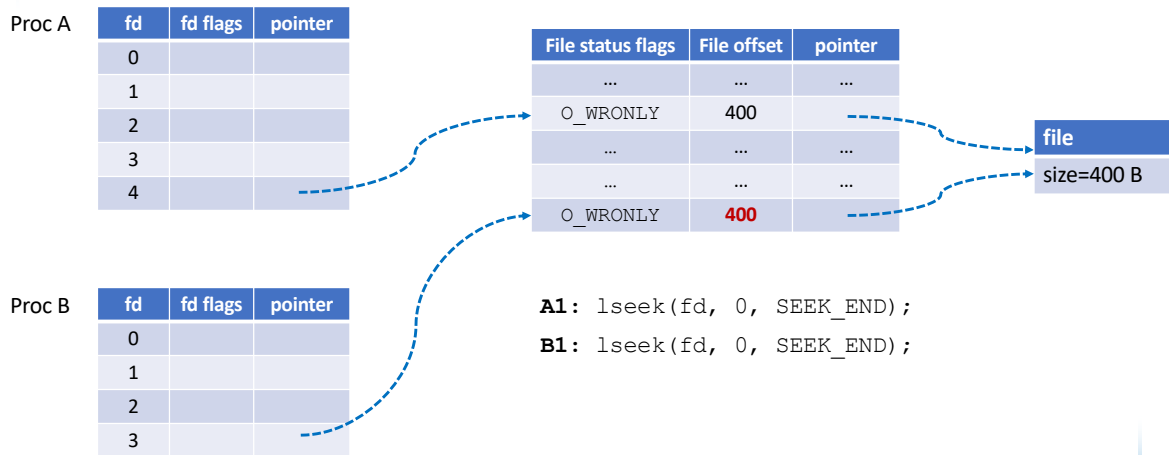
Condivisione di file tra processi

Due processi possono aprire lo stesso file contemporaneamente



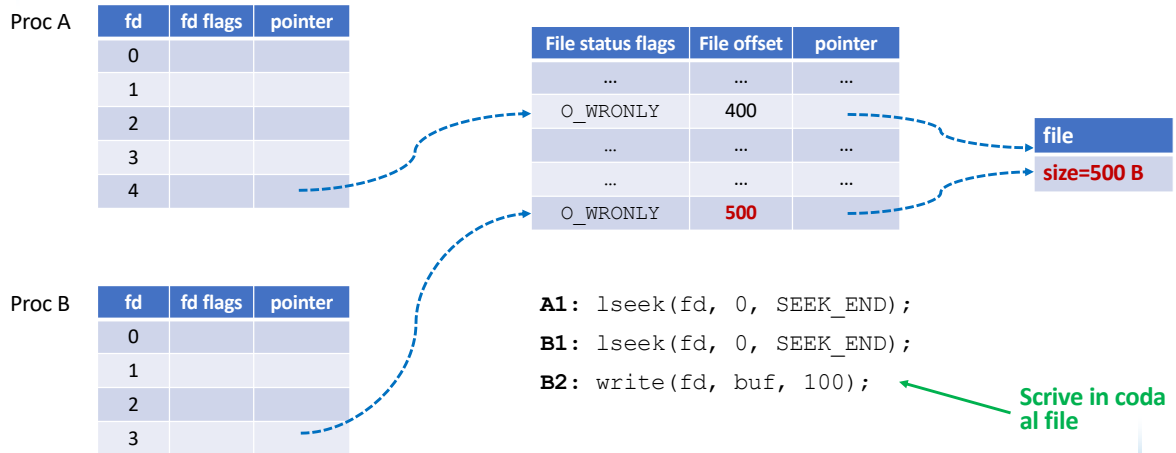
Condivisione di file tra processi

Due processi possono aprire lo stesso file contemporaneamente



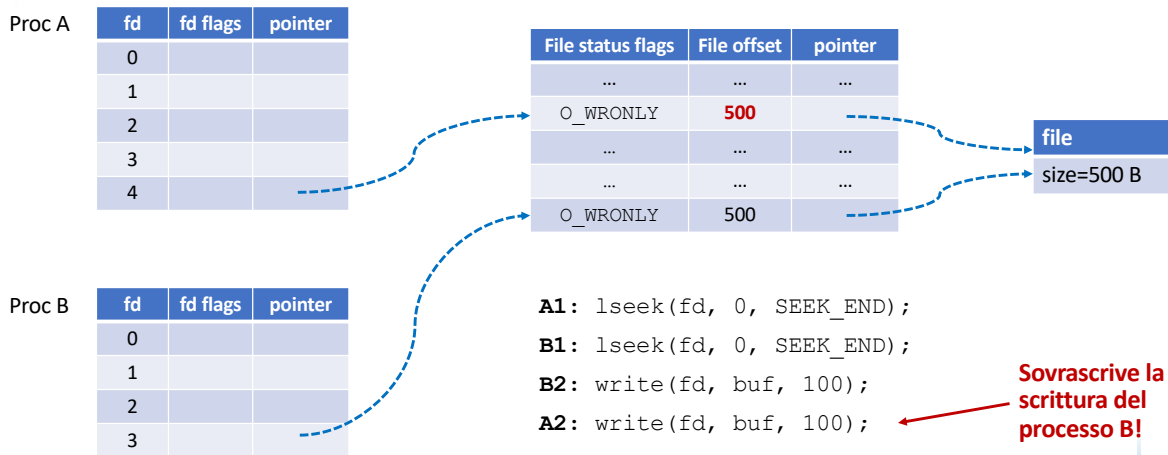
Condivisione di file tra processi

Due processi possono aprire lo stesso file contemporaneamente



Condivisione di file tra processi

Due processi possono aprire lo stesso file contemporaneamente



Unbuffered I/O

Ing. Giovanni Nardini - University of Pisa - All rights reserved

59

L'istruzione A2 sovrascrive ciò che aveva scritto l'istruzione B2, perché l'offset di partenza per il file descriptor del processo A era 400, mentre il processo B aveva già scritto 100 byte a partire da 400.

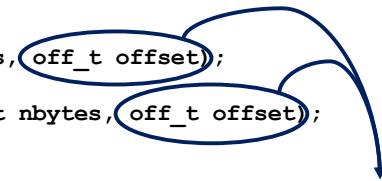
Operazioni atomiche

- **Problema:** un'operazione che richiede più chiamate di funzioni separate non è atomica
 - Il kernel può interrompere l'esecuzione in un punto intermedio qualsiasi

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

```
ssize_t pread(int fd, void* buf, size_t nbytes, off_t offset);
```

```
ssize_t pwrite(int fd, const void* buf, size_t nbytes, off_t offset);
```



Specifica la posizione in cui leggere/scrivere

- Operazioni atomiche per eseguire seek + read/write
- Non modificano l'offset nella file table