

# File I/O

Laboratorio Sistemi Operativi

Giuseppe Salvi

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

# UNIX: Programmazione di Sistema

- Per utilizzare i servizi offerti da UNIX, quali creazione di file, duplicazione di processi e comunicazione tra processi, i programmi applicativi devono interagire con il sistema operativo
- Per far ciò devono usare un insieme di routine dette **system call**, che costituiscono l'interfaccia funzionale del programmatore col nucleo di UNIX
- Le system call sono simili alle routine di libreria C ma eseguono una chiamata di subroutine direttamente nel nucleo di UNIX

# UNIX: System Call

- Le system call costituiscono un “entry point” per il kernel
- Il programmatore chiama la funzione utilizzando la sintassi usuale delle funzioni C
  - **`int open(const char *path, int mode)`**
- La funzione invoca, nel modo opportuno, il servizio del sistema operativo
  - “salva” gli argomenti della system call ed un numero identificativo della system call stessa (in registri)
  - esegue una istruzione macchina trap, ...

# UNIX: Funzioni di libreria

- Le funzioni di libreria forniscono servizi di utilità generale al programmatore
- Non sono entry point del kernel, anche se possono fare uso di system call per realizzare il proprio servizio
- Es:
  - *printf* può utilizzare la system call *write* per stampare
  - *strcpy* (string copy) e *atoi* (convert ASCII to integer) non coinvolgono il sistema operativo
- Possono essere sostituite con altre funzioni che realizzano lo stesso compito (in generale non possibile per le system call)

# Ricapitolando...(1)

- Il SO fornisce i servizi di base
  - Aprire un file, leggere un file, allocare memoria
- Servizi forniti tramite delle routine del kernel
- La libreria offre delle funzioni omonime che ci permettono di chiamare le **system call** come se fossero delle funzioni C
  - Il programma utente chiama la funzione C della libreria
  - La funzione C della libreria chiama il kernel
- Semplificando: system call= funzione C

# Ricapitolando...(2)

- La libreria contiene oltre alle system call anche altre funzioni di utilità generale
- **printf**
  - chiama la system call **write** per stampare i messaggi su video (file standard output)
- **malloc**
  - **sbrk** è la system call che alloca un certo numero di byte
- **strcpy**
  - non chiama nessuna system call

# Gestione dei file

# Gestione dei file

- Le system call per la gestione dei file permettono di manipolare
  - file regolari
  - directory
  - file speciali
- Tra i file speciali
  - link simbolici
  - dispositivi (terminali, stampanti)
  - meccanismi di IPC (pipe e socket)



# Chiamate di sistema di I/O

- Le `system call` descritte nella prima parte realizzano le operazioni di base per la gestione dei file
  - `open()`
  - `read()`
  - `write()`
  - `lseek()`
  - `close()`
- Spesso a tali funzioni ci si riferisce come I/O non bufferizzato
  - Ogni `read` o `write` invoca una `system call` del kernel

# Sequenze tipiche di operazioni con file

```
int fd; /* dichiara un descrittore di file */
...
fd = open(fileName, ...); /* apre un file ; fd è il
    descrittore */
if (fd == -1) { /* gestisce l'errore */
    ...
};
read(fd, ...); /* legge dal file */
...
write(fd, ...); /* scrive nel file */
...
lseek(fd, ...); /* si sposta all'interno del file */
...
close(fd); /* chiude il file, liberando il descrittore */
...
unlink(fileName); /* rimuove il file */
```

# Chiamate di sistema di I/O

- I file aperti sono gestiti dal kernel mediante **descrittori di file**
  - Un descrittore di file è un intero non negativo
- I descrittori di file possono variare da 0 a OPEN\_MAX (costante POSIX per il numero massimo di file aperti per processo. NB: non più usata in Linux)
  - Le prime versioni dei sistemi Unix avevano un limite superiore di 19, consentendo un massimo di 20 file aperti per processo. Molti sistemi hanno incrementato tale limite a 63
  - Con FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8 e Solaris 10, il limite è praticamente infinito, vincolato solo dalla quantità di memoria del sistema e dalla dimensione di un intero

# Chiamate di sistema di I/O

- Alla richiesta di aprire un file esistente o di creare un nuovo file il kernel ritorna un descrittore di file al processo chiamante
- Quando si vuole leggere o scrivere su un file si passa come argomento a `read` e `write` il descrittore ritornato da `open`
- Per convenzione
  - il descrittore `0` viene associato allo `standard input`
  - Il descrittore `1` allo `standard output`
  - Il descrittore `2` allo `standard error`
- Per conformità allo standard Posix, i numeri `0`, `1` e `2` possono essere sostituiti dalle costanti `STDIN_FILENO`, `STDOUT_FILENO` e `STDERR_FILENO`, definite nell'header `<unistd.h>`

# Chiamata di sistema open

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

```
int open(const char *path, int oflag, /*mode_t mode*/...);
```

- Funzione per aprire o creare file
  - `path` è il nome del file da creare o aprire
  - Il terzo argomento viene utilizzato solo quando si crea un file
  - Ritorna il descrittore del file, `-1` in caso di errore

# Chiamata di sistema open

- oflag
  - può assumere diversi valori (`<fcntl.h>`)
    - `O_RDONLY` apri solo in lettura
    - `O_WRONLY` apri solo in scrittura
    - `O_RDWR` apri in lettura e scrittura
  - Solo una delle precedenti costanti può essere specificata

# Chiamata di sistema open

- Le seguenti costanti sono opzionali:
  - **O\_APPEND** esegue un'aggiunta alla fine del file per ciascuna scrittura
  - **O\_CREAT** crea il file se non esiste
  - **O\_EXCL** se utilizzato insieme a O\_CREAT, ritorna un errore se il file esiste (e la creazione è un'operazione atomica)

# Chiamata di sistema open

- **O\_TRUNC** se il file esiste, ed è aperto con successo per sola scrittura o per lettura-scrittura, lo tronca a lunghezza zero
- **O\_NOCTTY** se path è un terminal device, non lo rende il terminale di controllo del processo
- **O\_NONBLOCK** se path è una FIFO, un file a blocchi o a caratteri, apre in maniera non bloccante, sia in lettura sia in scrittura



# Chiamata di sistema open

- **mode** definisce i bit di permesso di accesso ai file

mode	Permesso
S_IRUSR	Lettura utente
S_IWUSR	Scrittura utente
S_IXUSR	Esecuzione utente
S_IRGRP	Lettura gruppo
S_IWGRP	Scrittura gruppo
S_IXGRP	Esecuzione gruppo
S_IROTH	Lettura altri
S_IWOTH	Scrittura altri
S_IXOTH	Esecuzione altri

# Chiamata di sistema creat

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

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

- Funzione per creare file
  - creat apre un file in sola scrittura
    - Ritorna -1 in caso di errore
  - Essa è equivalente a:

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

# Chiamata di sistema creat

- Nelle precedenti versioni di Unix, il secondo argomento della `open` poteva essere solo 0,1 o 2
  - Non c'era modo di aprire un file che non esisteva
  - Fu introdotta `creat` per la creazione di nuovi file
  - Con le opzioni `O_CREAT` e `O_TRUNC`, la `open` è in grado di aprire nuovi file
    - Non c'è più la necessità di avere la funzione `creat`
  - Il problema della `creat` è che un file è aperto solo in scrittura
    - Prima della nuova `open`, dovendo creare un file temporaneo da scrivere e poi leggere, la sequenza di chiamate era `creat`, `close` e `open`
  - Per avere un file temporaneo per leggere e scrivere si può invocare, invece

```
open(path, O_RDWR | O_CREAT | O_TRUNC, mode);
```

# Chiamata di sistema close

```
#include <unistd.h>  
int close(int fildes);
```

- Chiude un file
  - ritorna -1 in caso di errore
  - quando un processo termina, tutti i file aperti vengono automaticamente chiusi

# Chiamata di sistema lseek

- Ad ogni file aperto è associato un valore intero non negativo, detto **offset corrente del file**, che misura il numero di byte dall'inizio del file

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

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

- Ritorna il nuovo offset, -1 in caso di errore
- Quando il file viene aperto l'offset viene inizializzato a zero, a meno che non si specifichi l'opzione **O\_APPEND**

# Chiamata di sistema lseek

- L'argomento **whence** può assumere i seguenti valori:
  - **SEEK\_SET** l'offset viene posto a offset byte dall'inizio del file
  - **SEEK\_CUR** viene aggiunto offset all'offset corrente
  - **SEEK\_END** l'offset viene posto alla fine del file, più offset

# Chiamata di sistema lseek

- Poiché una chiamata a `lseek` andata a buon fine restituisce il nuovo offset del file, per determinare l'offset corrente:
  - Si cercano zero byte dalla posizione corrente  
`off_t currpos;`  
`currpos = lseek(fd, 0, SEEK_CUR);`
  - Tale tecnica è usata anche per determinare se un file è in grado di essere “cercato”
    - Se il descrittore del file si riferisce a `pipe`, `fifo` o `socket`, `lseek` restituisce -1 e `errno` è impostata al valore `ESPIPE`

# Chiamata di sistema lseek

- L'offset di un file può essere più grande della dimensione corrente del file
  - La **write** successiva estende il file
  - Crea un buco
  - Qualsiasi byte nel file che non è stato scritto è letto come 0
  - Non è richiesto che ai buchi sia allocato un blocco su disco



# Esempio

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "apue.h"
char buf1[] = "abcdefghij";
char buf2[] = "ABCDEFGHIJ";
int main(void) {
    int fd;

    if ( (fd = creat("file.hole", FILE_MODE)) < 0) err_sys("creat error");

    if (write(fd, buf1, 10) != 10) err_sys("buf1 write error");
    /* offset ora = 10 */

    if (lseek(fd, 40, SEEK_SET) == -1) err_sys("lseek error");
    /* offset ora = 40 */

    if (write(fd, buf2, 10) != 10) err_sys("buf2 write error");
    /* offset ora = 50 */
    exit(0);
}
```

# Esempio

```
$ ./a.out
```

```
$ ls -l file.hole
```

```
-rw-r--r--  1  giusal   50   Jan 21 10:34 file.hole
```

```
$ od -c file.hole
```

```
0000000  a  b  c  d  e  f  g  h  i  j  \0  \0  \0  \0  \0  \0
0000020  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000040  \0 \0 \0 \0 \0 \0 \0 \0  A  B  C  D  E  F  G  H
0000060  I  J
0000062
```

- od elenca il contenuto di un file
  - Il flag `-c` dice di stampare il contenuto come caratteri
  - Il numero a sette cifre all'inizio di ogni riga è l'offset espresso in ottale

# Chiamata di sistema read

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

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

- Legge dal file **fildes**, **nbytes** byte in **buf**, a partire dalla posizione corrente
  - Aggiorna la posizione corrente
  - Ritorna il numero di byte effettivamente letti, 0 se alla fine del file, -1 in caso di errore

# Chiamata di sistema read

- Ci sono diversi casi in cui **read** legge un numero di byte minore di quanto richiesto:
  - Se è raggiunta la fine di un file (regolare) prima che sia letta la quantità di byte richiesti
  - Quando si legge da un terminale. Normalmente, è letta una riga per volta
  - Quando si legge da una rete. Il buffering nella rete può causare la restituzione di una quantità inferiore a quella richiesta
  - Quando si legge da una pipe o FIFO. Se la pipe contiene un numero inferiore di byte rispetto a quanto richiesto, è restituito solo ciò che è disponibile
  - Quando si è interrotti da un segnale ed è stata letta una quantità parziale di dati

# Chiamata di sistema write

```
#include <unistd.h>  
ssize_t write(int fildes, const void *buf, size_t nbytes);
```

- Scrive nel file **fildes**, **nbytes** byte da **buf**, a partire dalla posizione corrente
  - Aggiorna la posizione corrente
  - Restituisce il numero di byte effettivamente scritti, o -1 in caso di errore

# Esempio (mycat)

```
#define BUFFS 4096
int main(void)
{
    int n;
    char buf[BUFFS];
    while ((n=read(STDIN_FILENO, buf, BUFFS)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            printf("write error");
    if (n < 0)
        printf("read error");
    exit(0);
}
```

# Condivisione di file

- Unix supporta la condivisione dei file aperti tra differenti processi
- Il kernel usa tre strutture dati per rappresentare un file aperto
  - Le relazioni tra essi determinano l'effetto che un processo ha su di un altro processo rispetto alla condivisione dei file

# Condivisione di file

1. Ogni processo ha un'entrata nella tabella dei processi
  - All'interno di ogni entrata della tabella dei processi c'è una tabella dei descrittori di file aperti. A ciascun descrittore sono associati
    - I flag del descrittore di file
    - Un puntatore ad un'entrata della tabella dei file
2. Il kernel mantiene una tabella di file per tutti i file aperti. Ogni entrata contiene
  - I flag dello stato del file (read, write, append...)
  - L'offset corrente del file
  - Un puntatore alla entrata della tabella dei v-node per il file



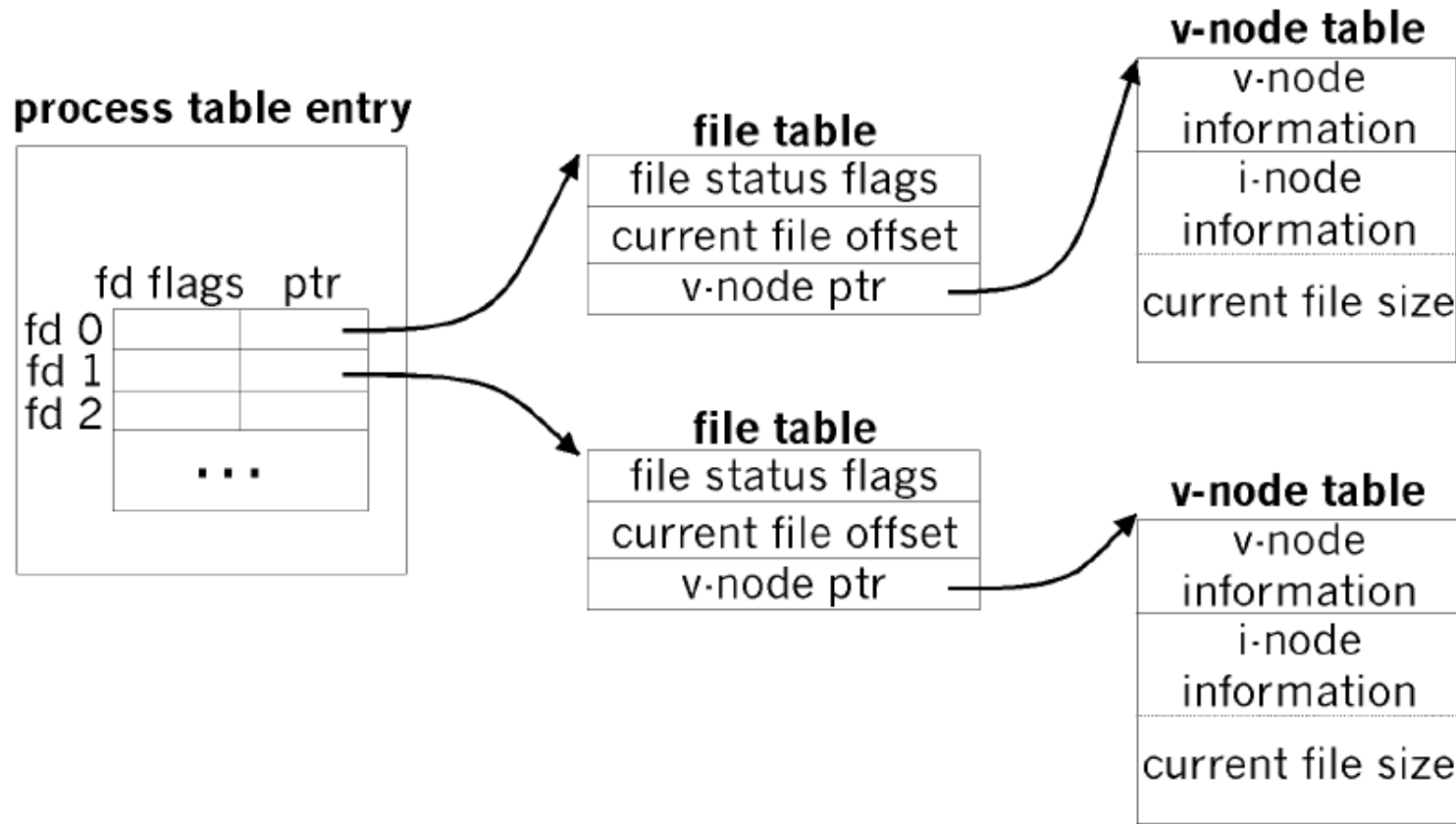
# Condivisione di file

3. Ogni file aperto (o device) ha una struttura v-node che contiene informazioni sul tipo del file e puntatori a funzioni che operano sul file
  - Il v-node contiene anche l'i-node per il file
  - Queste informazioni sono lette da disco quando il file è aperto
- N.B.: Linux non ha *v-node*. È usata, invece, una generica struttura *i-node*.

# Condivisione di file

- Esempio: vediamo le tre tabelle per un singolo processo con due differenti file aperti
  - Un file aperto sullo standard input (descrittore di file 0)
  - Un file aperto sullo standard output (descrittore di file1)

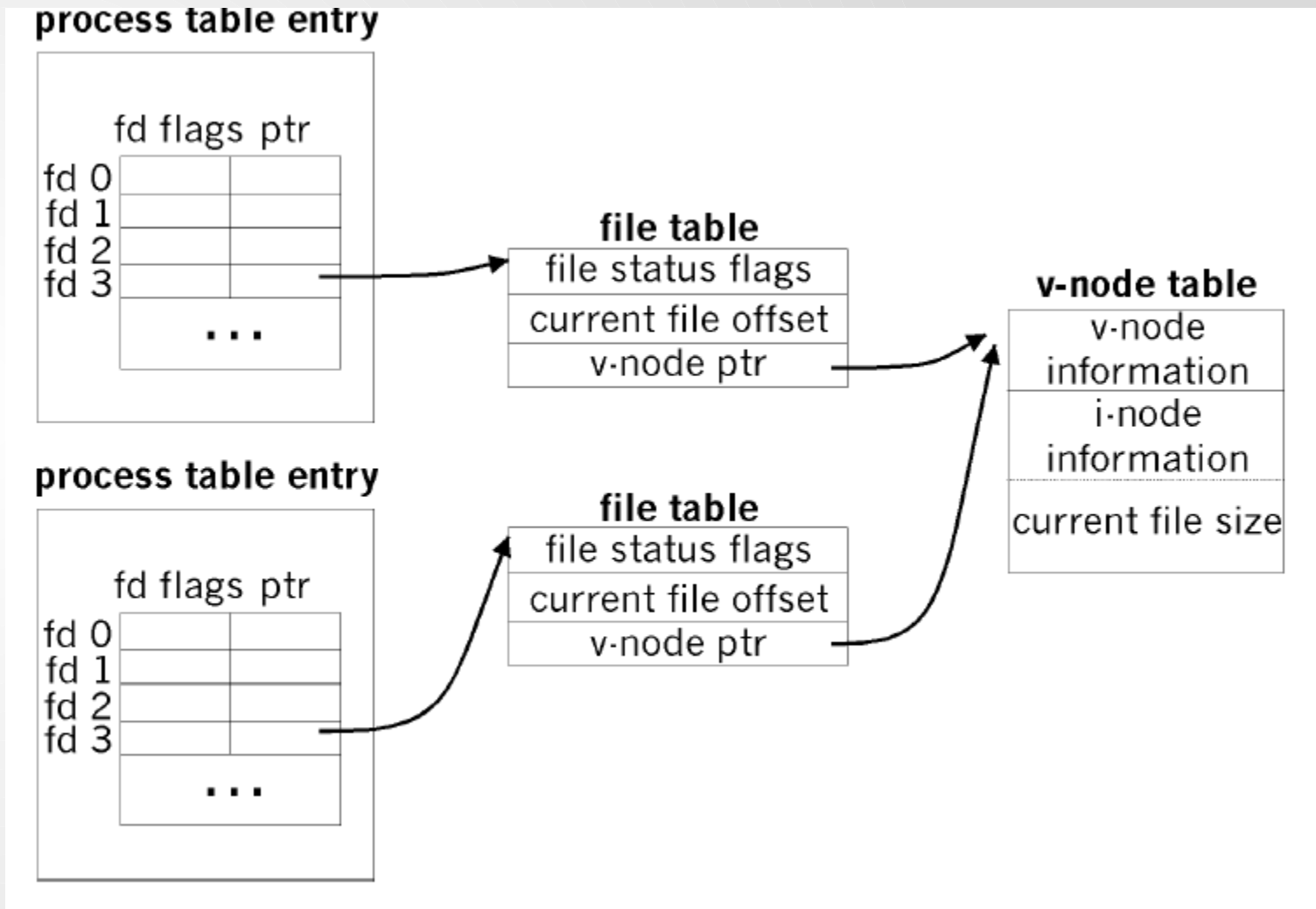
# Condivisione di file



# Condivisione di file

- Con due processi indipendenti che hanno lo stesso file aperto
  - Il primo processo ha il file aperto sul descrittore di file 3
  - Il secondo processo ha lo stesso file aperto sul descrittore 4
  - Ogni processo che apre il file ha la propria entrata nella tabella dei file, sebbene sia richiesta, per un dato file, solo una singola entrata della tabella dei v-node
    - La ragione per cui ciascun processo ha la propria entrata della tabella dei file è che ogni processo ha il proprio offset per il file

# Condivisione di file



# Condivisione di file

- Cosa accade quando un processo cerca di accedere ad un file?
  - Quando un processo accede ad un file mediante una **write**, l'elemento della tabella dei file relativo all'offset viene aggiornato e, se necessario viene aggiornato l'**i-node**
  - Se il file è aperto con **O\_APPEND**, è messo nella tabella dei file un flag corrispondente
  - Una chiamata ad **lseek** modifica solo l'offset corrente del file e non viene eseguita nessuna operazione di I/O
  - Se si chiede di posizionarsi alla fine del file, il valore corrente dell'offset nella tabella dei file viene preso dal campo della tavola di **i-node** che descrive la dimensione del file

# Operazioni atomiche

- Esempio: come aggiungere 100 byte alla fine di un file? (prime versioni di Unix)

```
if (lseek(fd, 0L, 2) < 0) /* posizionamento fine file */  
    err_sys ("lseek error");  
if (write(fd, buff, 100) != 100) /* scrittura */  
    err_sys ("write error");
```

- Cosa succede se due processi eseguono questa operazione su di uno stesso file?

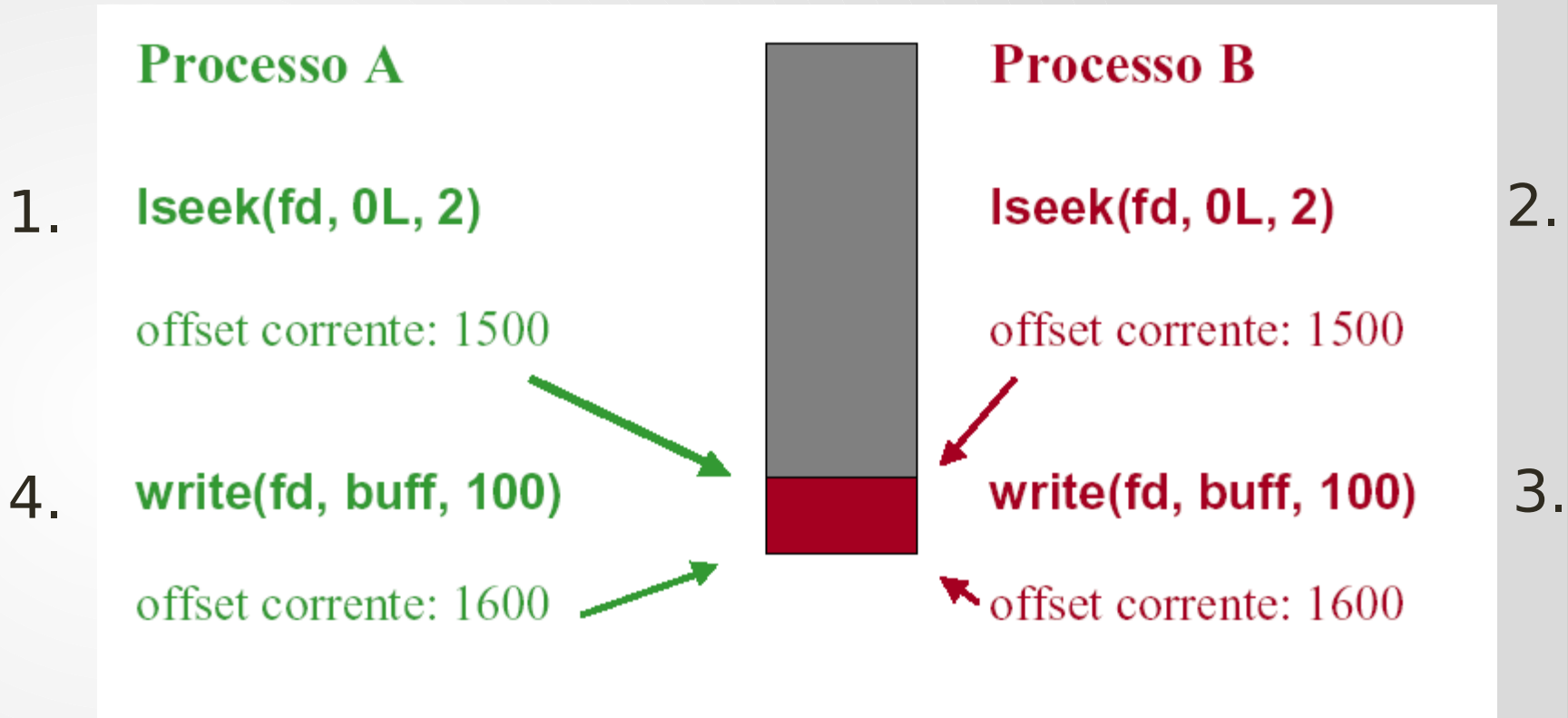
La L sta per **long** il motivo è che il secondo parametro di lseek è di tipo (primitivo) **off\_t** che è di tipo **long**

# Operazioni atomiche

- Questo tipo di operazione non comporta alcun problema per un unico processo
- Se più processi concorrenti impiegano questa tecnica per aggiungere dati alla fine di un file possono verificarsi dei problemi
- Supponiamo di avere due processi A e B che aggiungono byte alla fine di uno stesso file
  - Ognuno ha aperto il file senza l'opzione **O\_APPEND**
  - Supponiamo che l'attuale fine del file abbia offset pari a **1500**



# Operazioni atomiche (cont.)



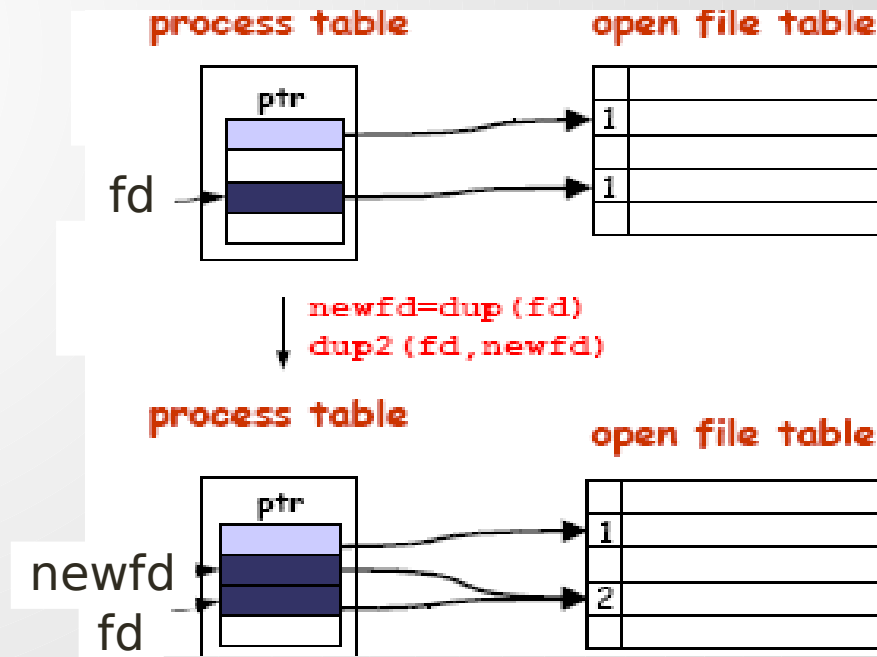
- Il processo **A** ha sovrascritto quello che ha scritto il processo **B**

# Operazioni atomiche (cont.)

- Perché Il processo **A** ha sovrascritto quello che ha scritto **B**?
  - L'operazione "posizionati alla fine del file e scrivi" richiede due azioni distinte (operazione non atomica)
  - Una qualsiasi operazione che richieda più di una chiamata a funzione può essere interrotta
  - Il modo per eseguire questa operazione in maniera atomica è di utilizzare il flag **O\_APPEND** quando si apre il file
    - Il kernel posiziona l'offset alla fine del file corrente prima di ogni write

# Duplicare i descrittori di file

- Le system call `dup()` e `dup2()` permettono di duplicare un file descriptor
  - Creano un (nuovo) file descriptor che punta alla stessa entry della tabella dei file aperti



# Le funzioni dup e dup2

```
#include <unistd.h>
int dup(int filedes);
int dup2(int filedes,int filedes2);
```

- **dup()**: trova il più piccolo descrittore non utilizzato e lo fa riferire alla stessa file descriptor entry (nella tabella dei file aperti) a cui fa riferimento **filedes**
- **dup2()**: il secondo argomento **filedes2** è il nuovo descrittore. Se **filedes2** è attualmente attivo, lo chiude e quindi lo fa riferire allo stesso file a cui fa riferimento **filedes**
- Nota: il descrittore di file originale e quello copiato condividono lo stesso puntatore interno al file e le stesse modalità di accesso
- Ritornano il nuovo descrittore, se hanno successo; e -1 altrimenti

# Esempio: duplicare i descrittori di file

```
#include <stdio.h>
#include <fcntl.h>
int main (void) {
int fd1, fd2, fd3;
fd1 = open ("test.txt", O_CREAT | O_RDWR, 0600);
printf ("fd1 = %d\n", fd1);
write (fd1, "Cosa sta", 8);
fd2 = dup (fd1); /* Effettua una copia di fd1 */
printf ("fd2 = %d\n", fd2);
write (fd2, " accadendo", 10);
close (0);      /* Chiude lo standard input */
fd3 = dup (fd1); /* Effettua un'altra copia di fd1 */
printf ("fd3 = %d\n", fd3);
write (0, " al contenuto", 13);
dup2 (3, 2);    /* Duplica il canale 3 sul canale 2 */
write (2, "?\n", 2);
return 0;
}
```

# Esempio: duplicare i descrittori di file

```
$ ./mydup
```

```
fd1 = 3
```

```
fd2 = 4
```

```
fd3 = 0
```

```
$ cat test.txt
```

Cosa sta accadendo al contenuto?

```
$
```

# Gestione degli errori: perror()

- Una system call ritorna -1 se fallisce
- Per gestire gli errori originati dalle system call, i due principali ingredienti da utilizzare sono:
  - **errno** variabile globale che contiene il codice numerico dell'ultimo errore generato da una system call
  - **perror()** subroutine che mostra una descrizione “testuale” dell'ultimo errore generato dall'invocazione di una system call

# Gestione degli errori: errno

- Variabile globale `errno`
  - inizializzata a 0
  - se si verifica un errore dovuto ad una system call, ad `errno` è assegnato un codice numerico corrispondente
- `errno.h` contiene codici di errore predefiniti.

Esempio:

```
#define EPERM 1 /* Operation not permitted */
#define ENOENT 2 /* No such file or directory */
#define ESRCH 3 /* No such process */
#define EINTR 4 /* Interrupted system call */
#define EIO 5 /* I/O error */
```

- Una system call che fallisce sovrascrive il valore di `errno`. Una eseguita con successo ... dipende (meglio salvare l'errore, se serve)



# Gestione degli errori: perror()

## void perror (char \*str)

- Mostra la stringa `str`, seguita da “:” e da una stringa che descrive il valore corrente di `errno` (chiusa da newline)
- Se non ci sono errori da riportare, viene mostrata la stringa `Error 0` (o, in alcuni sistemi, *Success*)
- Non è una system call, ma una routine di libreria
- Per accedere alla variabile `errno` ed invocare `perror()` occorre includere il file `errno.h`
- I programmi dovrebbero controllare se il valore ritornato da una system call è -1 e, in questo caso, invocare `perror()` per una descrizione dell'errore

# Esempio: showErrno.c

```
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
int main(void) {
int fd;
/* Apre un file inesistente per causare un errore */
fd = open ("nonexist.txt", O_RDONLY);
if (fd == -1) { /* fd=-1 => si è verificato un errore */
printf ("errno = %d\n", errno);
perror ("main");
}
fd = open ("/", O_WRONLY); /* Forza un errore diverso */
if (fd == -1) {
printf ("errno = %d\n", errno);
perror ("main");
}
/* continua ... */
```

# Gestione degli errori: perror()

```
/* Esegue una system call con successo */  
errno = 0; /* Reset manuale variabile di  
errore */  
perror ("main");  
return 0;  
}  
$ showErrno  
errno = 2  
main: No such file or directory  
errno = 21  
main: Is a directory  
main: Success  
$
```

# Esempio 1: copia di un file per carattere

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

```
...
```

```
int main(){  
    char c;  
    int in, out;  
    in = open("file.in",O_RDONLY);  
    out = open("file.out",O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);  
    while(read(in,&c,1)==1)  
        write(out,&c,1);  
    exit(0);  
}
```

# Esempio: copia di un file

```
/* mycopy src trg: crea una copia del file src e la chiama trg */
#include <stdio.h>
#include <sys/file.h>
#define BUFFSIZE 8192

int main(int argc, char *argv[]) {
    int fdSource; /* file descriptor per il file origine */
    int fdTarget; /* file descriptor per il file copia*/
    int n;
    char buf[BUFFSIZE]; /* buffer di transizione */
    fdSource = open(argv[1], O_RDONLY);
    fdTarget = open(argv[2], O_WRONLY | O_CREAT, 0600);

    /* copia il file sorgente sul target a blocchi di BUFFSIZE byte */
    while ( (n = read(fdSource, buf, BUFFSIZE)) > 0)
        if (write(fdTarget, buf, n) != n) {
            perror("write error");
            exit(1);
        }
}
```

# Esercizio 1

Scrivere un programma C “scrivi.c” che utilizzando la funzione primitiva “write”, scriva in un file “alfabeto.txt” la seguente stringa di caratteri:

“ABCDEFGHIJKLMNOPQRSTUVWXYZ”

# Esercizio 2

Scrivere un programma C “leggi.c” che utilizzando la funzione primitiva “read”, legga il contenuto del file “alfabeto.txt” e lo stampi sullo standard output.

# Esercizio 3

Usando la primitiva `lseek`, modificare il programma C `leggi.c` in modo da leggere e stampare su standard output solo la stringa `"ABFGMNRSZ"` contenuta nel file `"alfabeto.txt"`



# Esercizio 4

Si realizzi un programma “invertiByte.c” che stampi sullo standard output il contenuto di un file (passato come argomento) alla rovescia cioè a partire dall’ultimo carattere fino ad arrivare al primo.