

File I/O

File I/O

Gran parte delle operazioni su file in ambiente UNIX possono essere eseguite utilizzando solo cinque funzioni:

open

read

write

lseek

close

Descrittori di file

I file aperti sono gestiti dal kernel mediante **descrittori di file**.

Un descrittore di file è un intero non negativo

Tipo	ordinario, directory, speciale?
Posizione	dove si trova?
Dimensione	quanto è grande?
Numero di links	quanti nomi ha?
Proprietario	chi lo possiede?
Permessi	chi può usarlo e come?
Creazione	quando è stato creato?
Modifica	quando è stato modificato più di recente?
Accesso	quando è stato l'accesso più recente?

Descrittori di file (cont.)

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**, **1** allo **standard output** e **2** allo **standard error**

I numeri **0**, **1** e **2** possono essere sostituiti dalle costanti

STDIN_FILENO **STDOUT_FILENO** **STDERR_FILENO**

definite nell'header **<unistd.h>**

Chiamata di sistema `open`

```
#include <sys/types.h>
#include <sys/stat.h>
#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 per **creare**

Ritorna il descrittore del file, `-1` in caso di errore

Chiamata di sistema `open` (cont.)

oflag può assumere diversi valori (definiti in `<fcntl.h>`)

O_RDONLY

apri solo in lettura

O_WRONLY

apri solo in scrittura

O_RDWR

apri in lettura e scrittura

Chiamata di sistema open (cont.)

Solo una delle precedenti costanti può essere specificata, con una combinazione **OR** di:

O_APPEND esegue un *append* alla fine del file per ciascuna write

O_CREAT crea il file se non esiste

O_EXCL se utilizzato insieme a **O_CREAT**, ritorna un errore se il file esiste

Chiamata di sistema **open** (cont.)

Solo una delle precedenti **costanti** può essere specificata, con una combinazione **OR** di:

O_TRUNC se il file esiste, 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

O_SYNC attende che **write** sia **completata**

Chiamata di sistema **open** (cont.)

mode definisce i bit di permesso di accesso ai file

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 <sys/types.h>
#include <sys/stat.h>
#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);
```

Per avere un file temporaneo per leggere e scrivere si può

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

Chiamata di sistema close

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

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 **current file offset**, che misura il numero di byte dall'inizio del file

Quando il file viene aperto l'offset viene inizializzato a zero, a meno che non si specifichi l'opzione **O_APPEND**

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek (int fd, off_t offset, int whence);
```

Ritorna il nuovo offset, -1 in caso di errore

Chiamata di sistema lseek (cont.)

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

Esempio

```
#include <sys/types.h>
#include "ourhdr.h"

int main(void)
{
    if (lseek(STDIN_FILENO, 0, SEEK_CUR) == -1)
        printf("cannot seek\n");
    else
        printf("seek OK\n");
    exit(0);
}
```

Esempio

```
#include <sys/types.h>
#include "ourhdr.h"
```

```
int main(void)
{
    if (lseek(STDIN_FILENO, 0, SEEK_CUR) == -1)
        printf("cannot seek\n");
    else
        printf("seek OK\n");
    exit(0);
}
```

```
$ a.out < /etc/motd
seek OK
$ cat < /etc/motd | a.out
cannot seek
$ a.out < /var/spool/cron/FIFO
cannot seek
```

Esempio

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "ourhdr.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 now = 10 */

        if (lseek(fd, 40, SEEK_SET) == -1) err_sys("lseek error");

        /* offset now = 40 */

        if (write(fd, buf2, 10) != 10) err_sys("buf2 write error");

        /* offset now = 50 */

        exit(0);
}
```

Chiamata di sistema read

```
#include <unistd.h>
ssize_t read(int fildes, void *buff, 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, -1 in caso di errore

Chiamata di sistema write

```
#include <unistd.h>
ssize_t write(int fildes, const void *buff, 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

```
#include    "ourhdr.h"

#define     BUFFS  8192

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

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

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

    exit(0);
}
```

Efficienza dell'I/O

BUFFSIZE	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	# loops
1	23.8	397.9	423.4	1468802
2	12.3	202.0	215.2	734401
4	6.1	100.6	107.2	367201
8	3.0	50.7	54.0	183601
16	1.5	25.3	27.0	91801
32	0.7	12.8	13.7	45901
64	0.3	6.6	7.0	22950
128	0.2	3.3	3.6	11475
256	0.1	1.8	1.9	5738
512	0.0	1.0	1.1	2869
1024	0.0	0.6	0.6	1435
2048	0.0	0.4	0.4	718
4096	0.0	0.4	0.4	359
8192	0.0	0.3	0.3	180
16384	0.0	0.3	0.3	90
32768	0.0	0.3	0.3	45
65536	0.0	0.3	0.3	23
131072	0.0	0.3	0.3	12

Standard output ridiretto su **/dev/null**,
Berkeley file system, blocchi da 8192

M. R. Guarascino File I/O

Condivisione di file

Il kernel utilizza tre **strutture dati** per la gestione dell' I/O:

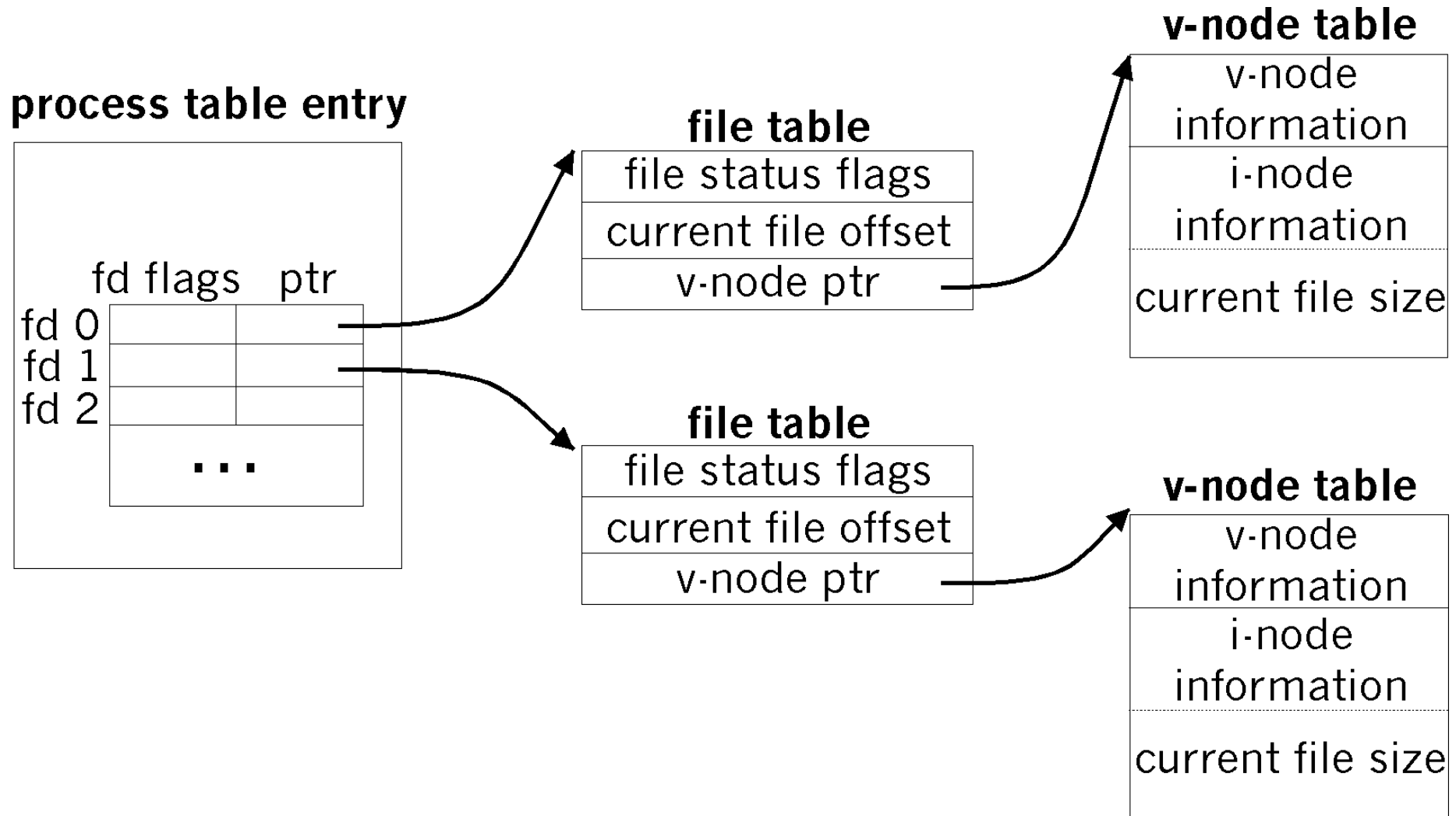
1. Ciascun processo ha un elemento nella **tabella dei processi**.

Tale elemento è un "vettore" di descrittori di file aperti, ciascuno con un puntatore ad un elemento della **tabella dei file**

2. Il kernel possiede una **tabella** per ciascun **file aperto** con i flag di stato del file (lettura, scrittura, append,...), l'offset corrente ed un puntatore ad un elemento della **tabella dei v-node**

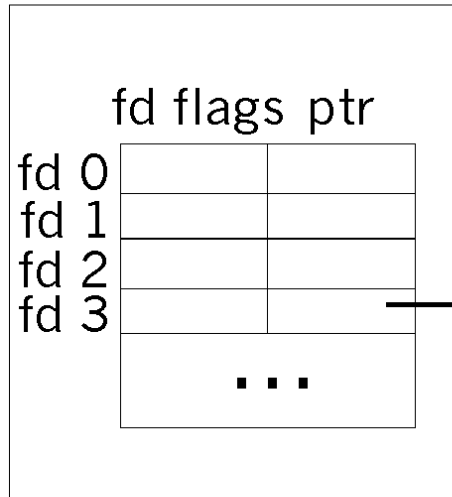
3. Ciascun file aperto ha una **struttura v-node**. Il v-node contiene informazioni sul tipo di file e sulle funzioni che operano su di esso.

Condivisione di file (cont.)

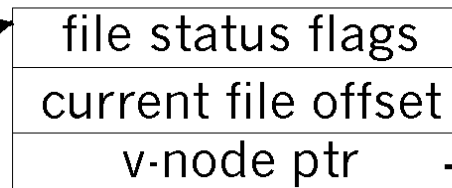


Condivisione di file (cont.)

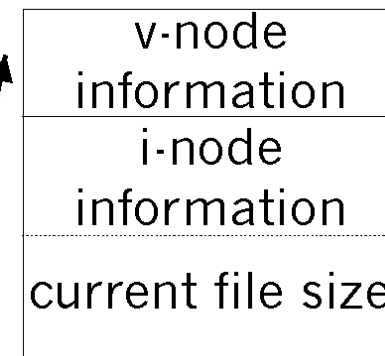
process table entry



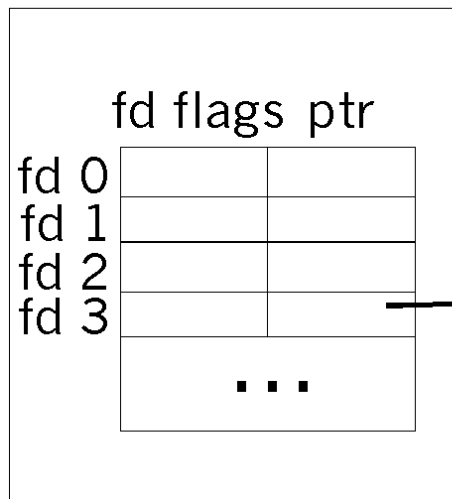
file table



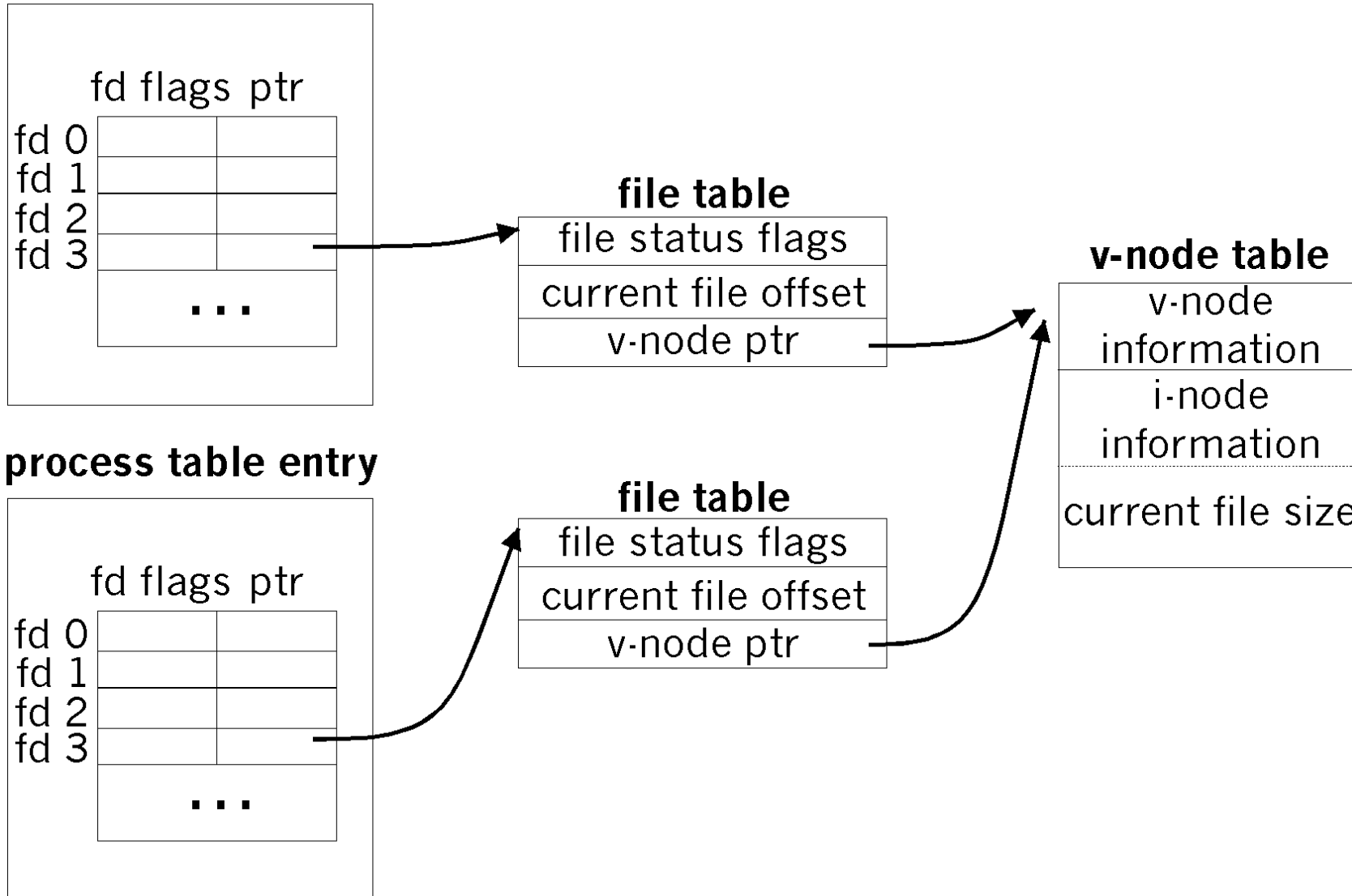
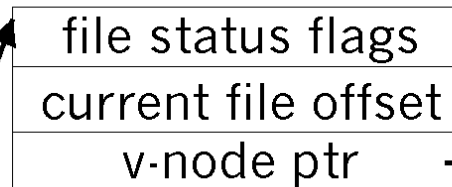
v-node table



process table entry



file table



Condivisione di file (cont.)

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**, un **flag** corrispondente è messo nella **tabella dei file**

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: aggiunta alla fine di un file

Come aggiungere 100 byte alla fine di un file?

```
if (lseek(fd, 0L, 2) < 0)          /* posizionamento  
                                   alla fine del 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**?

Processo A

`lseek(fd, 0L, 2)`

offset corrente: 1500

`write(fd, buff, 100)`

offset corrente: 1600



Processo B

`lseek(fd, 0L, 2)`

offset corrente: 1500

`write(fd, buff, 100)`

offset corrente: 1600

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

Perchè

Il **processo B** ha **sovrascritto** quello che ha scritto **A**?

L'operazione *posizionati alla fine del file e scrivi* richiede due azioni distinte

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

Operazioni atomiche (cont.)

Esempio: creazione di un file

Come aprire un file solo se non esiste?

```
if ( (fd = open (pathname, O_WRONLY)) < 0)
  if (errno == ENOENT) {
    if ( (fd = creat (pathname, mode)) < 0)
      err_sys ("creat error");
  } else
    err_sys ("open error");
```

Se il file viene creato da un altro processo tra le due chiamate, tutto ciò che viene scritto, viene cancellato da creat

Operazioni atomiche (cont.)

Per evitare questo problema bisogna rendere l'operazione di controllo di esistenza e creazione un'unica operazione

Se si apre un file con open e le opzioni **O_CREAT** e **O_EXCL**, l'apertura fallisce se il file esiste

Le **operazioni atomiche** si riferiscono ad operazioni composte da più azioni.

Se un'operazione è eseguita in maniera **atomica**, tutte le **azioni** sono portate **a compimento**

Funzioni dup e dup2

```
#include <unistd.h>  
int dup(int fildes);  
int dup2(int fildes, int fildes2);
```

Funzioni per duplicare i descrittori di file

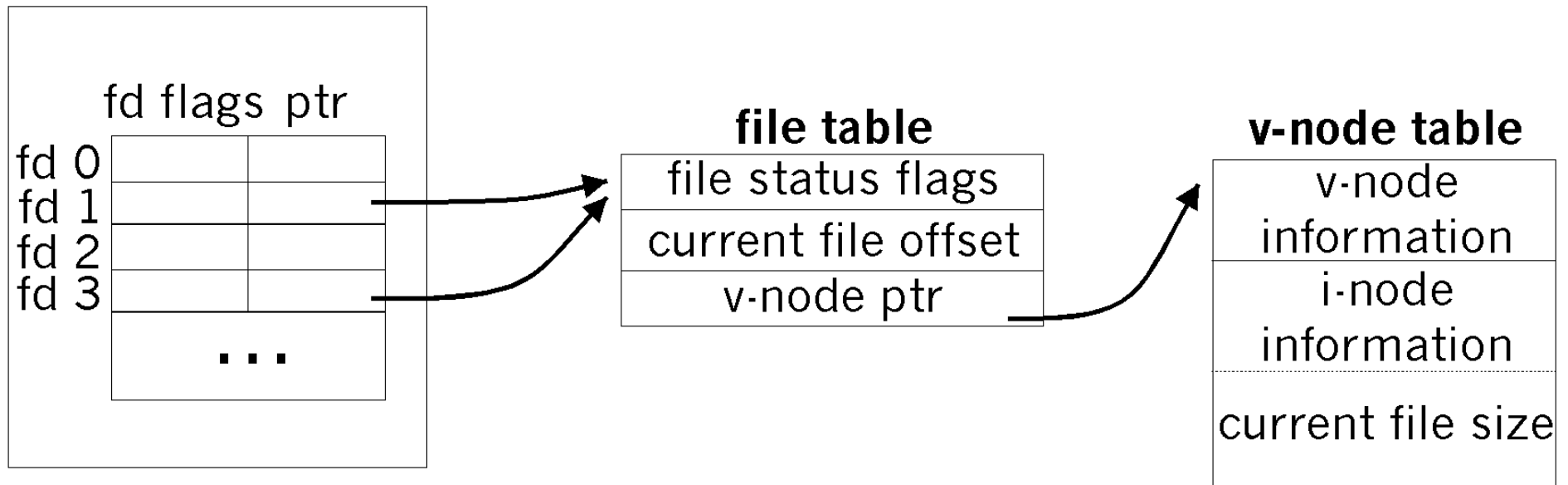
Ritornano un nuovo descrittore, o -1 in caso di errore

dup ritorna un descrittore con il valore più basso possibile

dup2 ritorna un descrittore con il valore di *fildes2*

Funzioni dup e dup2

process table entry



Sommario

Abbiamo visto le tradizionali funzioni di I/O di Unix
(*funzioni non bufferizzate*)

Abbiamo visto cosa siano le *funzioni atomiche*
e cosa succede quando più processi cercano
di accedere allo stesso file

Cose da leggere

Andare a guardare sul manuale di Unix i comandi di gestione dei file e delle directory