Laboratorio di sistemi operativi
A.A. 2010/2011
Gruppo 2
Gennaro Oliva
16
Input/Output

I/O buffering

- Esistono tre tipi di buffering:
  - Completo: le operazioni di I/O avvengono solo al riempimento del buffer; viene utilizzato, ad esempio, da printf per l'I/O su file
  - Buffering a linea: le operazioni di I/O avvengono non appena viene inserito nel buffer un carattere newline '\n' (o al termine del processo); viene utilizzato, ad esempio, da printf nell'output a terminale e da scanf nell'input da tastiera
  - Senza buffering: le operazioni di I/O avvengono immediatamente; viene utilizzato, ad esempio, da fprintf sullo standard error
- A differenza della libreria standard le system call implementano I/O non bufferizzato

System call per I/O UNIX

- Le system call fondamentali per le operazioni di input/output su file per sistemi UNIX sono 5

  open, read, write, lseek, close

- Le primitive read e write sono basilari e si utilizzano per diversi tipi di I/O (pipe, fifo, socket)
- Prima di qualsiasi operazione I/O (lettura, scrittura, scorrimento) su un file è necessario aprirlo
- In ogni processo, il kernel associa un intero non negativo unico chiamato file descriptor ad ogni file aperto

Canali I/O standard

- La shell UNIX assegna, per convenzione I seguenti file descriptor:
  - `0`: standard input
  - `1`: standard output
  - `2`: standard error
- In luogo degli interi, per migliorare la portabilità del codice vengono definite le seguenti le costanti simboliche:
  - STDIN_FILENO (`0`)
  - STDOUT_FILENO (`1`)
  - STDERR_FILENO (`2`)
- all'interno dell'header di sistema unistd.h

open

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

DESCRIPTION

- Consente di aprire un file esistente o di creare un file nel caso in cui non esista un file con il nome specificato
- Il parametro pathname contiene il pathname (assoluto o relativo) del file da aprire
- Il parametro flags permette di specificare molteplici opzioni di apertura combinandole mediante il simbolo “|”
- Il parametro mode è opzionale, definisce i permessi del file e viene utilizzato solo nel caso di creazione

open

- RETURN VALUE
  - La system call open restituisce in caso di successo il file descriptor assegnato dal kernel al file aperto oppure -1 in caso di errore
  - Il file descriptor restituito è il minimo intero che non sia già un file descriptor utilizzato dal processo
  - Il kernel riutilizza i file descriptor associati a file chiusi!

I flag di apertura

- Il parametro oflag deve necessariamente contenere esattamente uno dei seguenti parametri:
  - O_RDONLY: apre il file in sola lettura
  - O_WRONLY: apre il file in sola scrittura
  - O_RDWR: apre il file in lettura e scrittura
- In aggiunta è possibile specificare una o più delle seguenti opzioni:
  - O_APPEND – le operazioni di scrittura avvengono sempre a fine file
  - O_CREAT – crea il file se non esiste, richiede l'uso del terzo parametro
  - O_EXCL – genera un errore se il file esiste ed è stata specificata anche l'opzione O_CREAT
  - O_TRUNC – se il file esiste e se è stato aperto in scrittura o in lettura/scrittura viene cancellato all'apertura
  - O_SYNC – specifica che ogni chiamata write interrompe l'esecuzione del codice fino a quando non viene completa la scrittura sul dispositivo di memorizzazione
  - Le opzioni che si possono utilizzare dipendono dal tipo di filesystem su cui si sta chiedendo l'apertura del file

open: esempio

- Esempio di combinazione di più flag
- `f = open (“primo.txt”,O_WRONLY|O_APPEND|O_TRUNC);`
- Apre il file primo.txt in scrittura, soltanto se esiste, cancellandone il contenuto e posizionandosi in coda al file ad ogni operazione di scrittura

Il parametro mode

- Nella creazione di un nuovo file se viene specificato il parametro O_CREAT è necessario utilizzare il parametro mode per definire i permessi del file
- Il parametro mode consente di specificare ogni singolo bit dei nove che costituiscono i permessi del file attraverso una serie di costanti combinate con il simbolo “|”

Costanti per il parametro mode

- S_IRWXU: rwx per il proprietario
- S_IRUSR: r per il proprietario
- S_IWUSR: w per il proprietario
- S_IXUSR: x per il proprietario
- S_IRWXG: rwx per il gruppo
- S_IRGRP: r per il gruppo
- S_IWGRP: w per il gruppo
- S_IXGRP: x per il gruppo
- S_IRWXO: rwx per gli altri
- S_IROTH: r per gli altri
- S_IWOTH: w per gli altri
- S_IXOTH: x per gli altri

open: esempio

- Esempio di combinazione di più flag
- `f = open ( "primo.txt", O_WRONLY|O_APPEND|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);`
- Apre il file primo.txt in scrittura cancellandone il contenuto se esiste e posizionandosi in coda al file ad ogni operazione di scrittura
- Se il file non esiste lo crea con i permessi

  rw-r--r--

close

SYNOPSIS

```
#include <unistd.h>

int close(int fd);
```

DESCRIPTION

- Chiude il file associato al file descriptor fd

RETURN VALUE

- Restituisce `0` in caso di successo `-1` in caso di errore

offset

- Ad ogni file aperto è associato un intero, detto offset, che rappresenta la posizione in cui verrà effettuata la prossima operazione di I/O
- La posizione è la distanza misurata in byte dall'inizio del file
- Si può immaginare l'offset come la posizione della testina sull'hard disk o su un nastro magnetico
- La open inizializza sempre l'offset a zero
- Le operazioni di lettura e scrittura vengono effettuate a patire dall'offset corrente e ne incrementano del numero di byte di un numero pari a quelli letti o scritti
- È possibile modificare l’offset in modo arbitrario utilizzando la system call lseek

offset e O_APPEND

- Una open invocata in scrittura o in lettura-scrittura con l'opzione O_APPEND inizializza comunque l'offset a zero
- Le operazioni di lettura incrementano il valore dell'offset di un numero di byte pari al quelli letti
- Prima di ogni operazione di scrittura l'offset viene spostato a fine file

lseek

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

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

DESCRIPTION

- Sposta l'offset di un file aperto associato al file descriptor fd di nbytes a partire dalla posizione specificata nel parametro whence con i seguenti parametri:
- SEEK_SET: la posizione di riferimento è l'inizio del file
- SEEK_CUR: la posizione di riferimento è quella dell'offset corrente
- SEEK_END: la posizione di riferimento è la fine del file
- Il valore del parametro offset può essere sia positivo che negativo

lseek

DESCRIPTION (continua)

- La system call lseek non consente di specificare nessuno spostamento all'interno di un file che abbia come risultato un offset finale negativo (non posso andare prima dell'inzio di un file)
- Per contro l'offset del file può essere posizionato anche dopo la fine di file (ma questo non cambia la dimensione di un file)
- Se vengono scritti dati in un punto che sta oltre la fine del file viene generato un buco nel file e ogni lettura di byte all'interno del buco restituisce il byte nullo fino a quando non vengono scritti dati diversi all'interno del buco
- Un esempio di utilizzo di questa tecnica e il file che emula un disco di una macchina virtuale

lseek

- RETURN VALUE
  - In caso di successo fseek restituisce l'offset del file dopo il posizionamento
  - In caso di errore viene restituito il valore (off_t) -1
  - Quindi per conoscere l'offset corrente, è possibile utilizzare:

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

read

- SYNOPSIS

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

- DESCRIPTION
  - Prova a leggere fino a count byte a partire dall'offset corrente del file descriptor fd e memorizzarli all'interno dell'area di memoria che parte dall'indirizzo buf
  - Il buffer di destinazione nel prototipo è void * ovvero puntatore “generico” per consentirci di leggere qualsiasi tipo di dato
  - La read non fa altro che leggere la sequenza di byte (contigui) all'interno del file ed memorizzarla nell'area di memoria che parte dall’indirizzo indicato da buff

Problemi di compatibilità

- Attenzione: il C non esegue alcun controllo sulla compatibilità dei tipi di dato e non memorizza alcuna informazione sul tipo di dato salvato
- Possono verificarsi problemi di compatibilità tra i tipi su architetture non compatibili che rappresentano un determinato tipo con una diversa codifica (ad esempio little-endian e big-endian) o con un numero di byte diversi
- Scrivendo il file su un architettura e leggendolo su una non compatibile potrebbe generare errori di interpretazione ovvero assegnazioni di valori non corretti
- Il tipo char non ha problemi di compatibilità in quanto utilizza sempre la codifica ASCII

read

- RETURN VALUE
  - In caso di successo read restituisce il numero di byte effettivamente letti e incrementa l'offset del file di questo valore
  - Se viene la lettura a partire da un offset che va oltre la fine del file oppure viene specificato il valore `0` come count la read restituisce `0`
  - In caso di errore il valore di ritorno è -1
  - Il numero di byte effettivamente letti può essere inferiore al parametro count quando:
    - Il numero di byte presenti nel file a partire dall'offset corrente è inferiore a count
    - La lettura avviene da tastiera
    - La lettura avviene da un file descriptor che rappresenta uno strumento di comunicazione interprocesso quale un socket di rete una pipe o una FIFO
    - L'operazione viene interrotta da un segnale

Allocazione e scrittura

- Anche nel caso delle system call non viene effettuato alcun controllo sulla dimensione del buffer
- Se il buffer specificato come secondo parametro non è abbastanza grande per memorizzare il numero di byte specificati in count si possono modificare erroneamente altre aree di memoria
- Qualora queste ricadano al di fuori dello spazio di indirizzamento allocato per il processo in esecuzione, il kernel bloccherà il tentativo di accesso alla memoria terminando il programma

write

- SYNOPSIS

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

- DESCRIPTION
  - Prova a scrivere fino a count byte a partire dall'offset corrente del file descriptor fd e prendendoli all'interno dell'area di memoria che parte dall'indirizzo buf
  - Il buffer di origine nel prototipo è void * ovvero puntatore “generico” per consentirci di scrivere qualsiasi tipo di dato
  - La write non fa altro che scrivere la sequenza di byte (contigui) all'interno del file leggendoli dall'area di memoria che parte dall’indirizzo indicato da buff

write

- RETURN VALUE
  - In caso di successo write restituisce il numero di byte effettivamente scritti e incrementa l'offset del file di questo valore
  - In caso di errore il valore di ritorno è -1
  - Il numero di byte effettivamente scritti può essere inferiore al parametro count
  - È sempre necessario opportuno verificare il valore di ritorno delle system call

La variabile errno

- Molte system call restituiscono -1 in caso di errore un valore generico che non è indicativo dell'errore riscontrato
- Per fornire più informazioni sull'errore, lo standard C prevede l'utilizzo della variabile globale errno (error number)
- Le system call che restituiscono il valore -1 assegnano un valore alla variabile errno per consentirci di identificare il problema riscontrato
- Il valore che una system call può assegnare alla variabile errno in caso di errore è menzionato nella pagina di manuale della system call nella sotto sezione ERRORS

Costanti numeriche d'errore

- La lista dei valori possibili per la variabile errno, sotto forma di costanti simboliche è disponibile nella pagina di manuale errno(3)
- Alcuni esempi di costanti numeriche d'errore standard
- EACCES Permission denied
- ENOENT No such file or directory
- ENOSPC No space left on device
- ETIMEDOUT Connection timed out
- Per convertire le constanti in un messaggio d'errore si può utilizzare la funzione strerror mentre per visualizzare direttamente il messaggio d'errore in base al contenuto corrente di errno si può utilizzare la funzione perror

La funzione perror

- SYNOPSIS

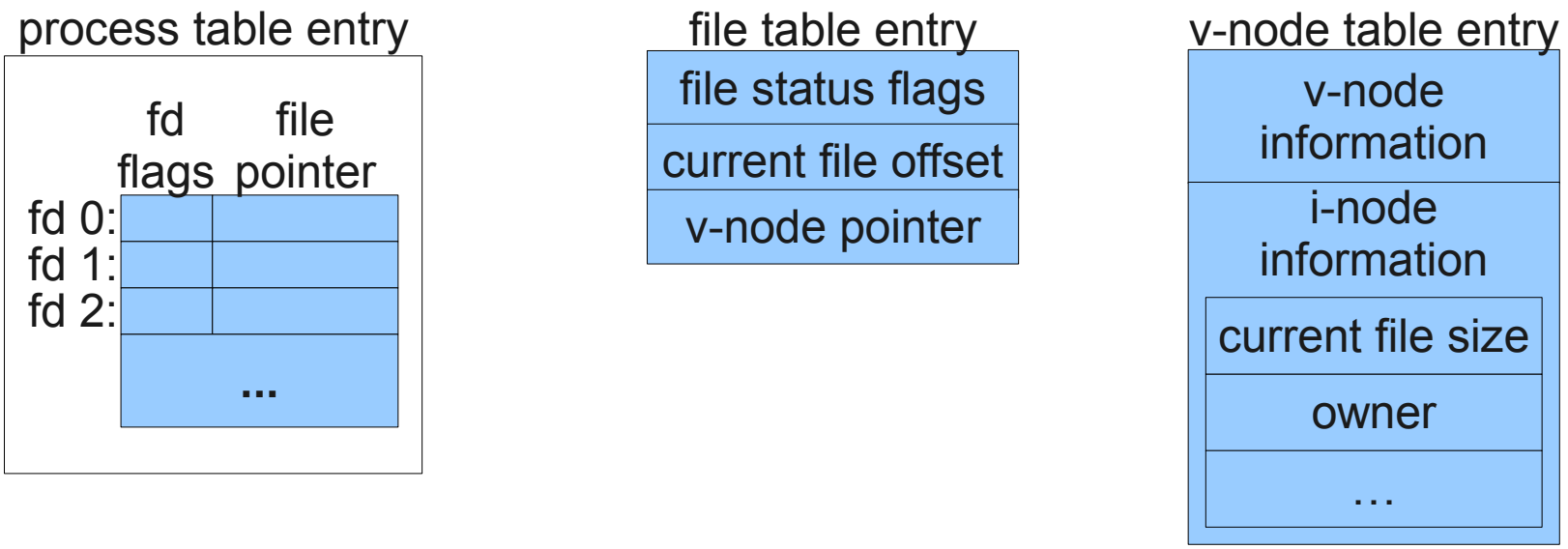
```
#include <stdio.h>

void perror(const char *s);
```

- DESCRIPTION
  - La funzione perror(const char *) stampa la stringa passata come parametro seguita da “:” e successivamente il messaggio d'errore corrispondente al valore corrente di errno
- Esempio d'uso:
-
```
if ((fd=open("data",O_RDONLY))<0)
        perror("open");
```
- Esempio di messaggio:
-
```
open: No such file or directory
```

La condivisione di file aperti

- Il sistema operativo UNIX supporta la condivisione di file aperti tra processi utilizzando 3 strutture dati
- Process table, file table, v-node table

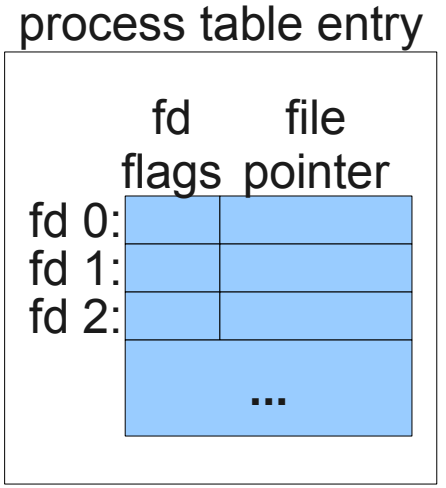


Dettagli implementativi

- La descrizione è puramente illustrativa e volutamente ignora dettagli implementativi quali:
- l'area di memoria che detiene le informazioni della tabella dei file aperti (spazio utente o kernel)
- le strutture dati utilizzate per memorizzare le tabelle (array, linked list, ...)

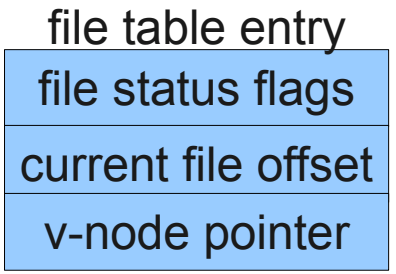
Process table

- La tabella dei processi contiene informazioni su tutti i processi in esecuzione sul sistema
- Ogni processo ha una corrispondente voce nella tabella all'interno della quale è memorizzato un vettore con i file descriptor aperti
- Ogni elemento del vettore contiene il file descriptor flag close-on-exec (che vedremo quando affronteremo la gestione dei processi) ed un puntatore ad una voce della tabella dei file



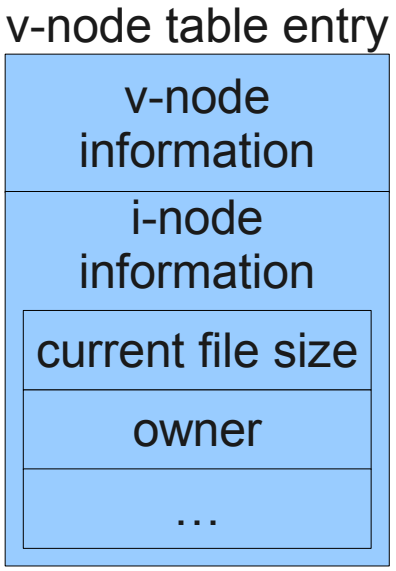
File table

- La tabella dei file contiene informazioni su tutti i file aperti
- In ogni elemento della tabella vengono memorizzati lo status flag (read,write,append,sync e nonblocking)
- L'offset corrente del file
- Un puntatore all'elemento corrispondente nella tabella dei v-node



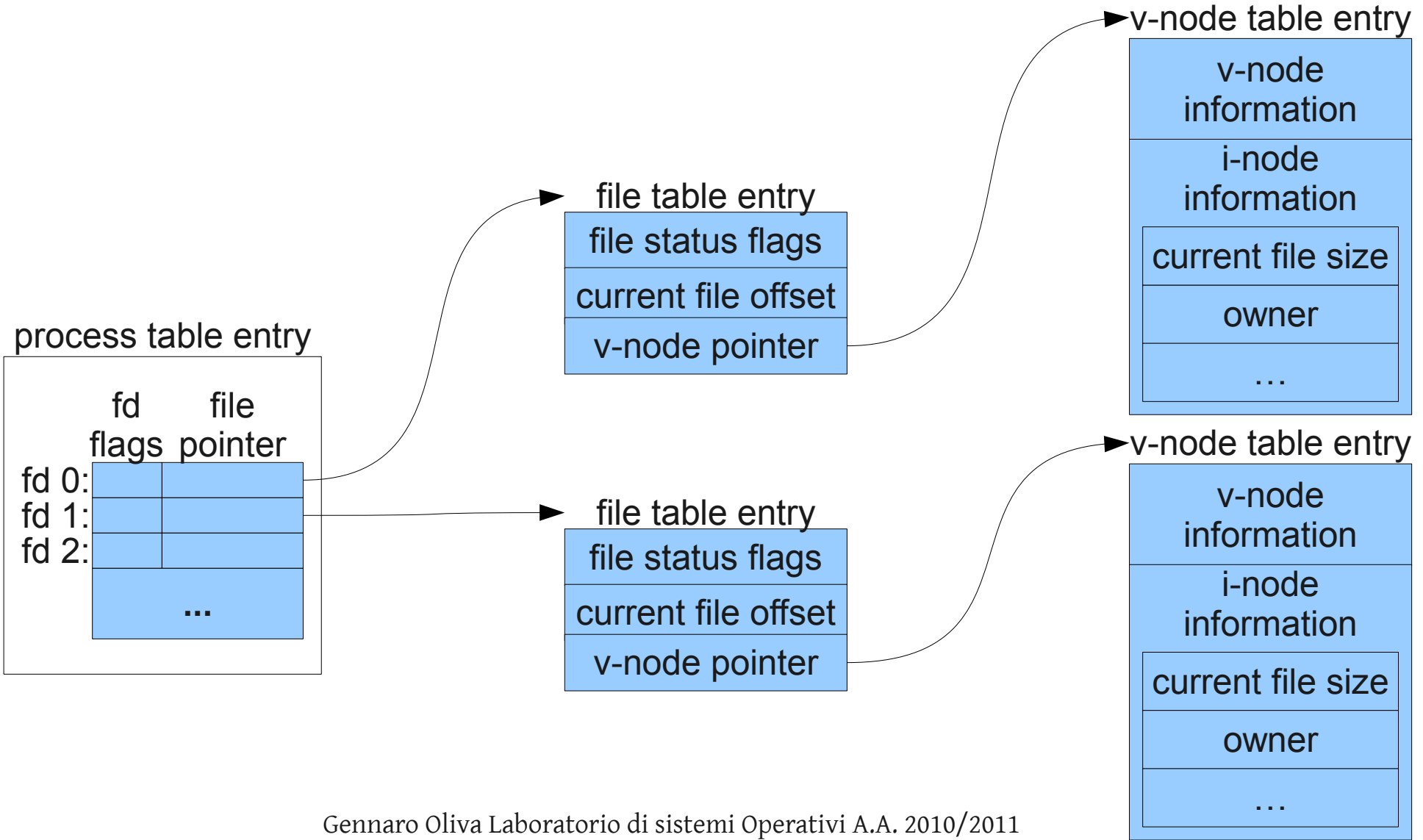
v-node table

- Nella tabella dei v-node ogni elemento contiene informazioni sul file quali l'i-node del file che viene letto dal disco quando il file viene aperto
- L'i-node contiene informazioni quali il proprietario del file, i permessi, la dimensione del file, gli indirizzi dei blocchi di disco che contengono il file



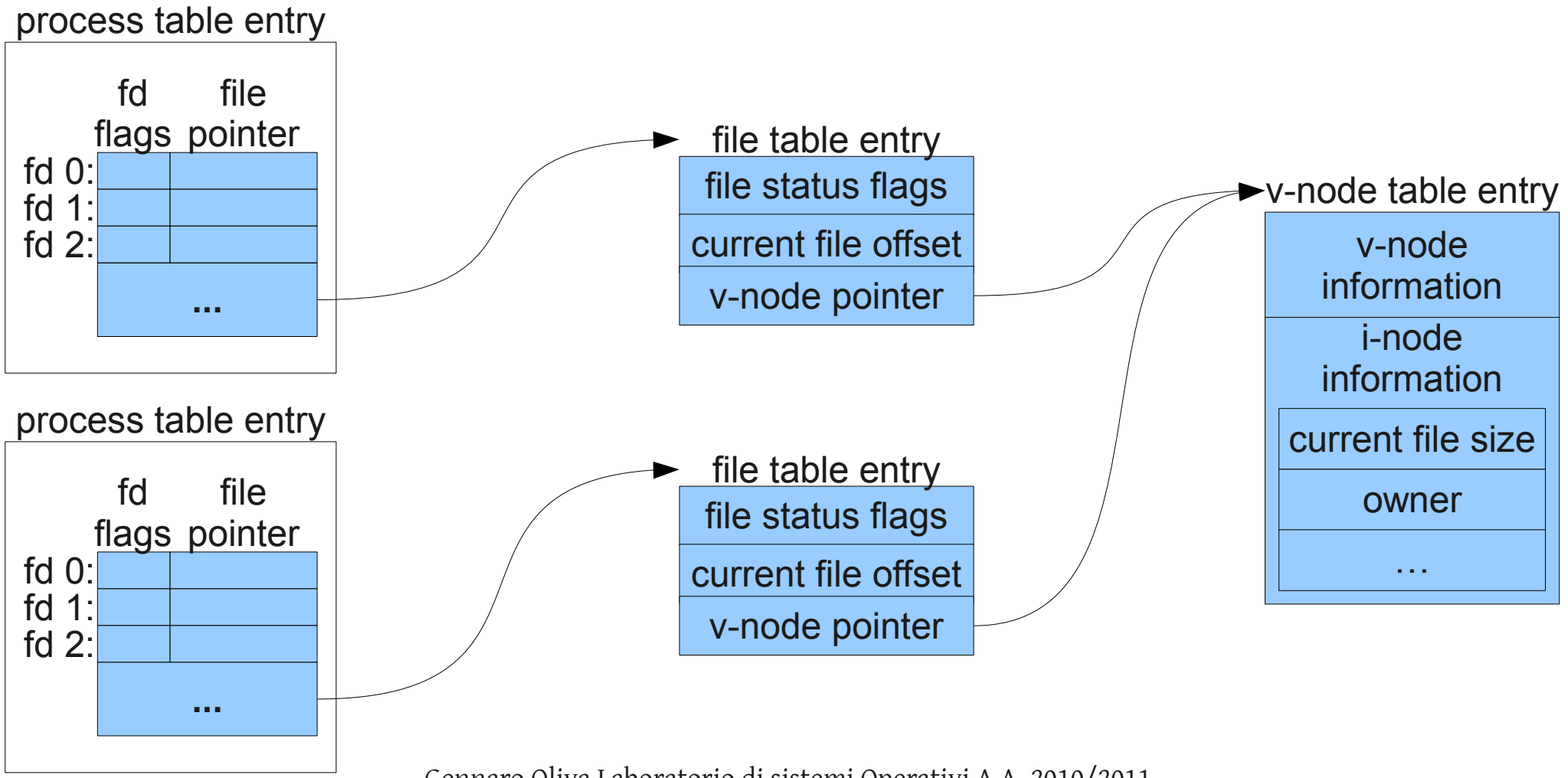
Relazione tra le tabelle

- La figura illustra un singolo processo che ha due diversi file aperti uno come standard input (fd `0`) ed uno sullo standard output (fd 1)



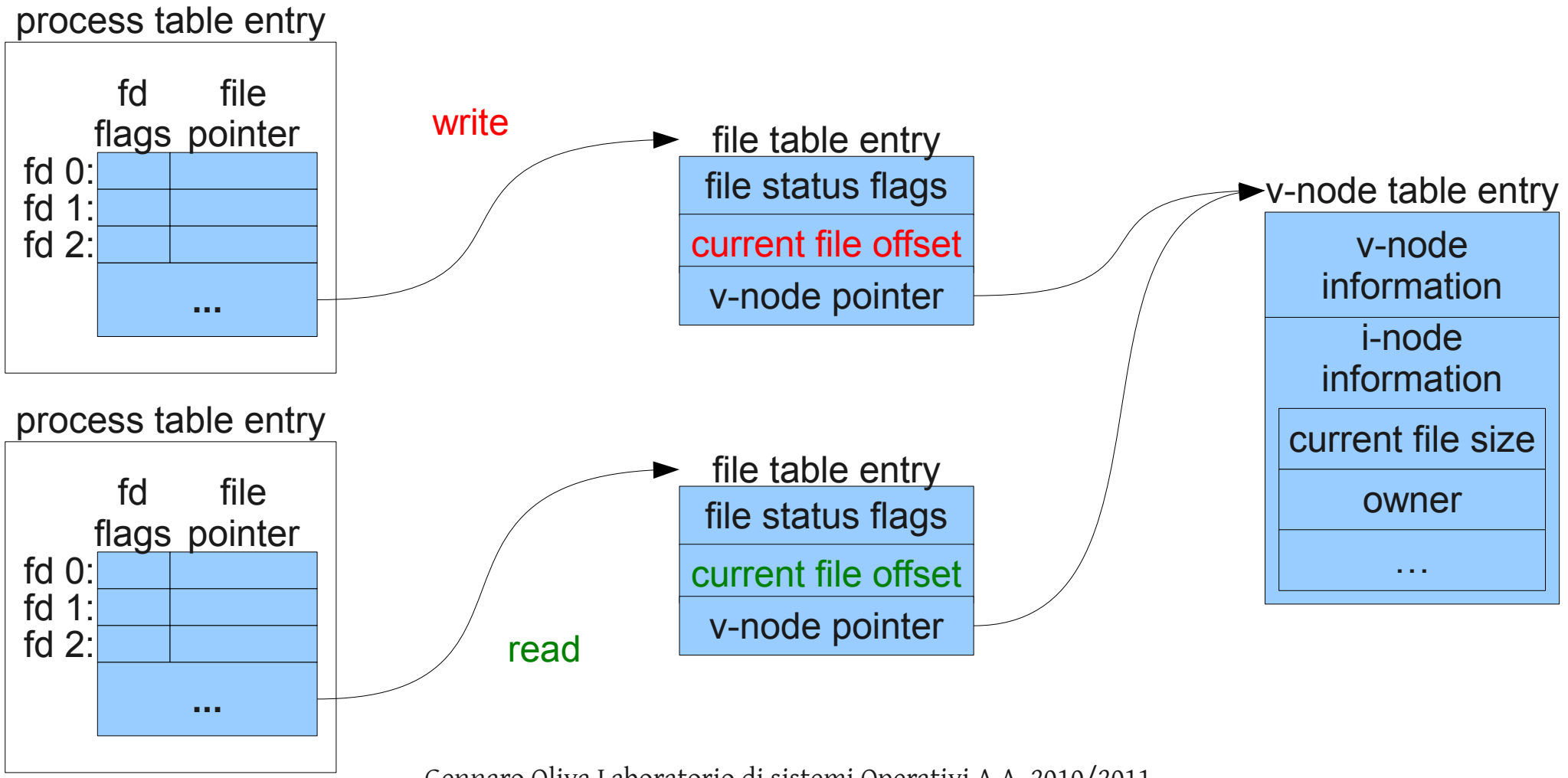
Relazione tra le tabelle

- La figura illustra due processo che aprono uno stesso file



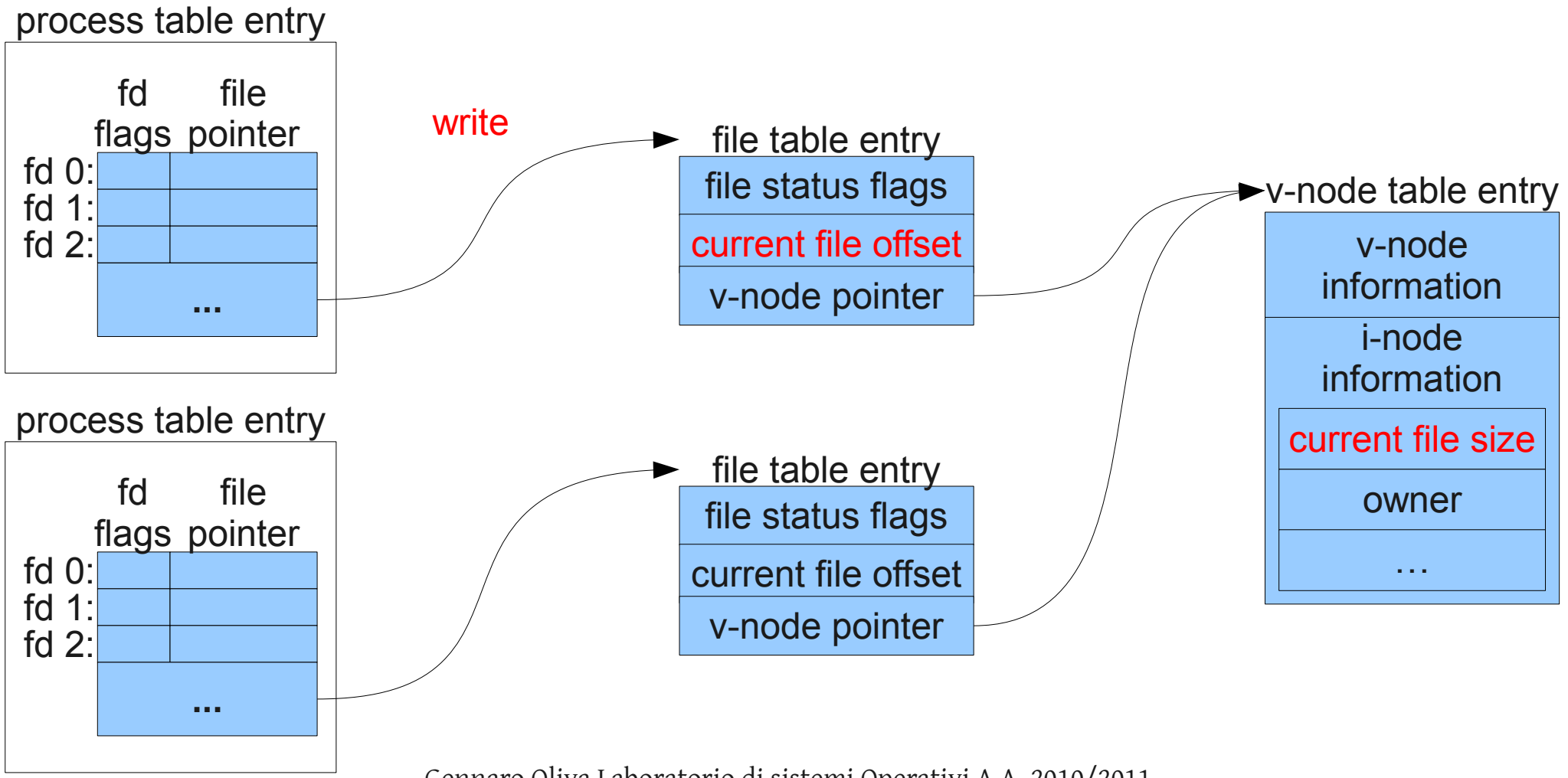
Relazione tra le tabelle

- Al termine di ogni operazione di I/O l'offset corrente nella file table viene incrementato del numero di byte scritti o letti ed in questo caso è distinto per ciascun processo



Relazione tra le tabelle

- Se le operazioni di scrittura modificano la dimensione del file l'i-node corrispondente viene aggiornato ed il dato è unico per tutti i processi



Processi concorrenti e file comuni

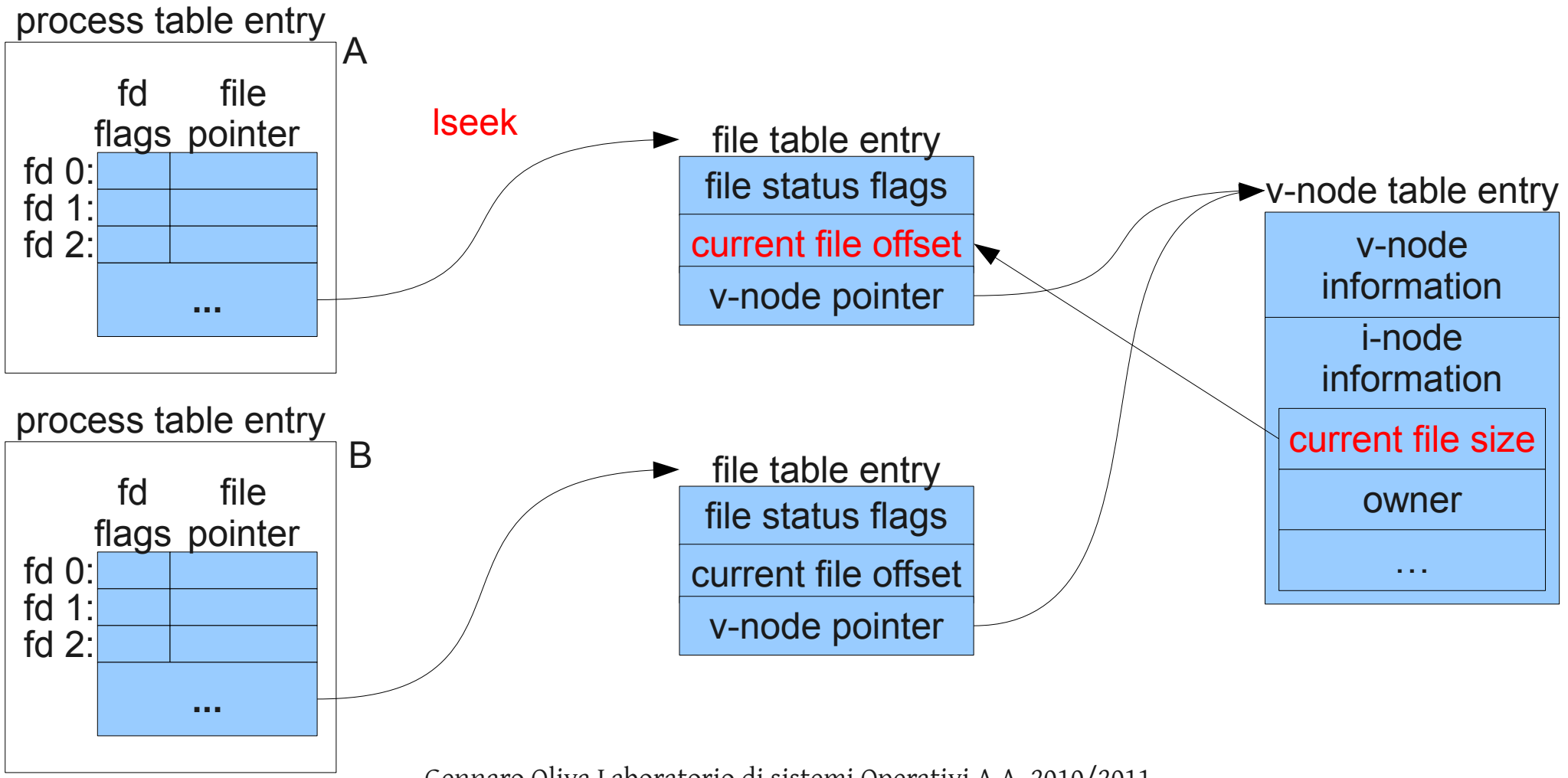
- Si consideri il caso di due o più processi che vogliono scrivere in coda ad un file
- Questo tipo di esigenza si verifica tipicamente dei file di log dei servizi attivi sul sistema
- Le vecchie versioni di UNIX non supportavano l'opzione O_APPEND per cui si doveva operare come segue

```
if (lseek(fd, 0, SEEK_END) < 0)
   perror("lseek error");
else if (write(fd, buf, 100) != 100)
   perror("write error");
```

- In un ambiente multitasking quale UNIX questo approccio è problematico

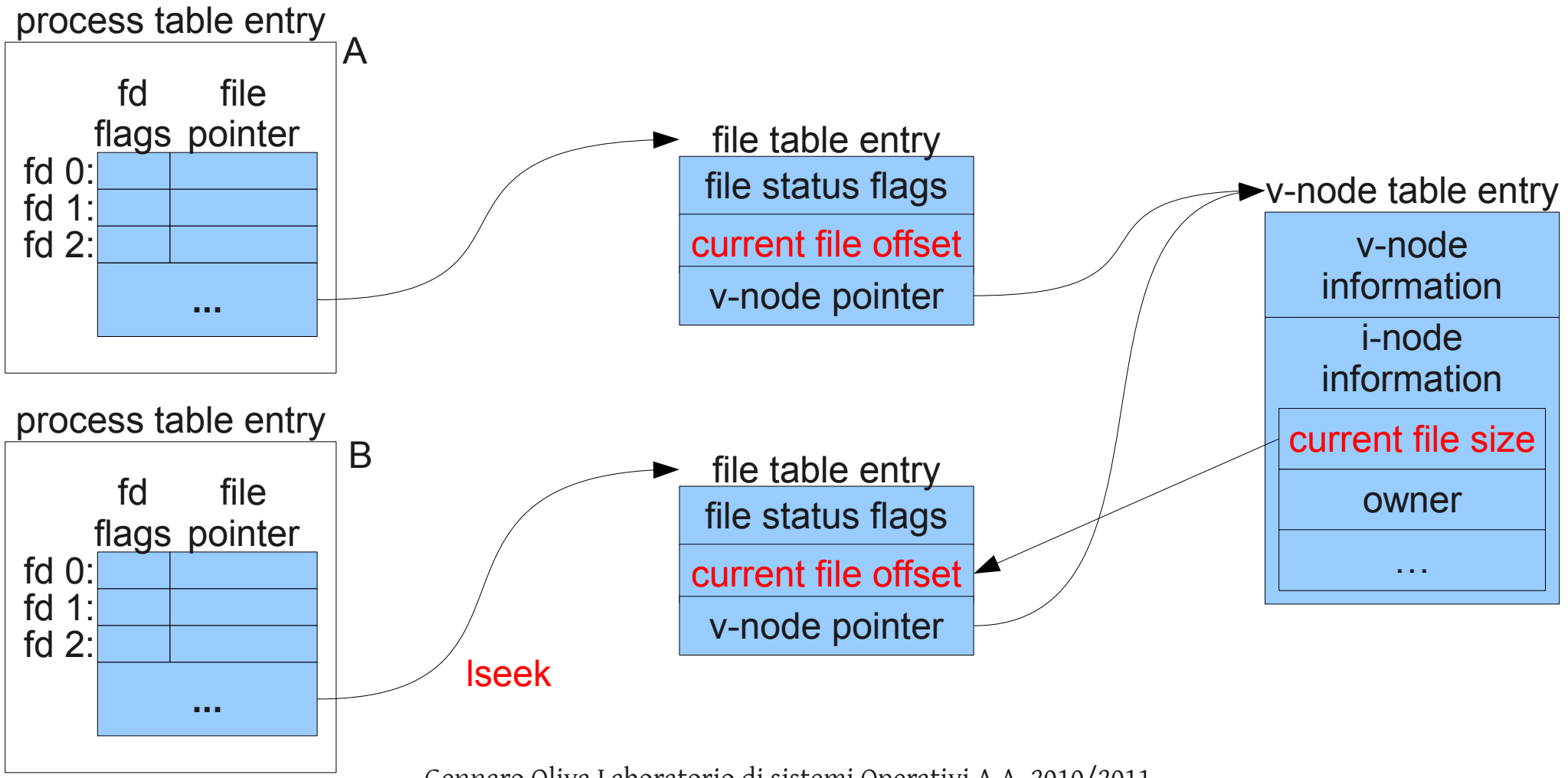
Processi concorrenti e file comuni

- Si supponga che un processo A, volendo scrivere in coda al file, effettui la chiamata ad lseek posizionando l'offset alla fine del file



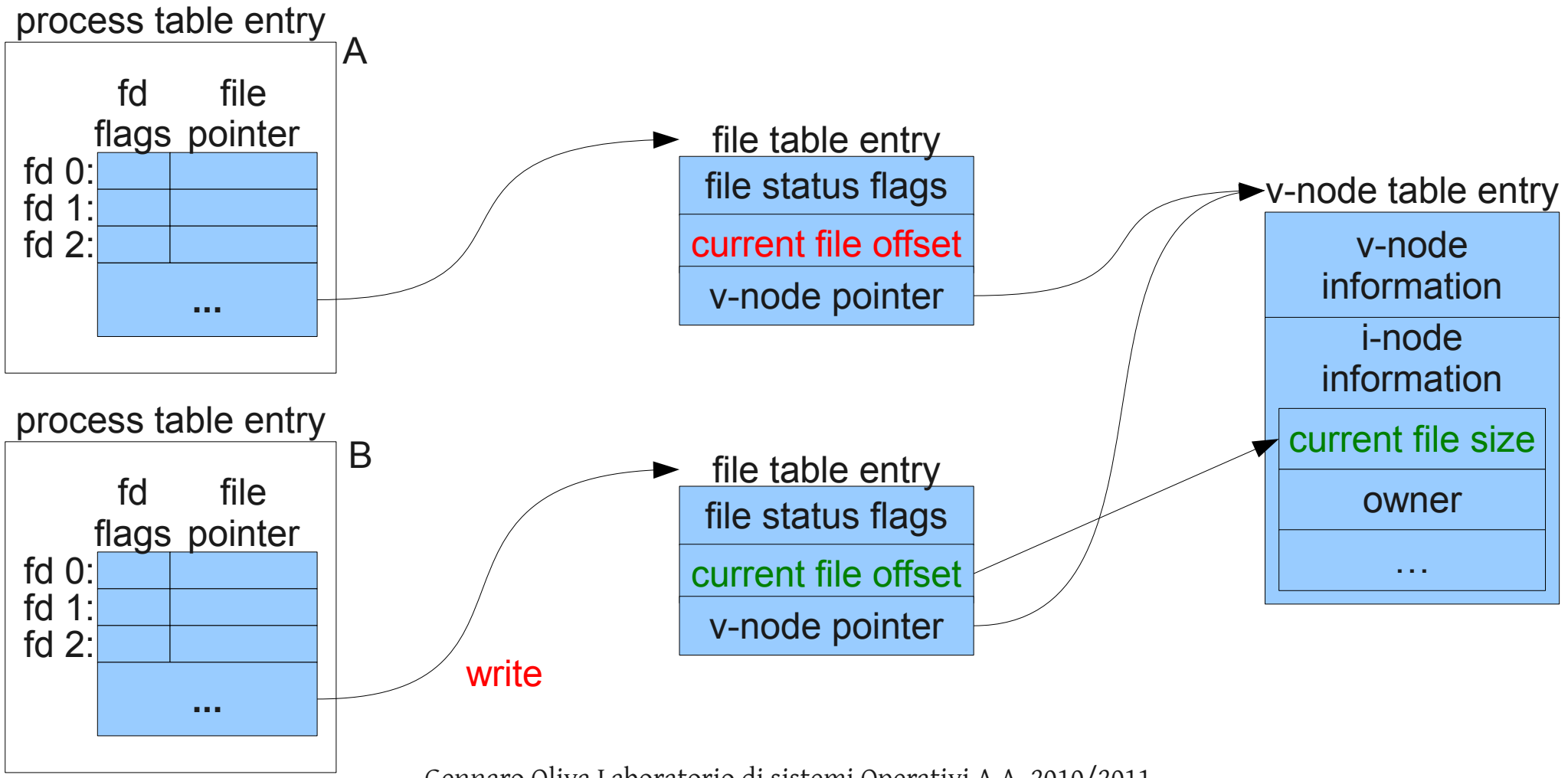
Processi concorrenti e file comuni

- Si supponga a questo punto che il kernel interrompa l'esecuzione del processo A e ponga in esecuzione un secondo processo B che tenta di fare la stessa operazione



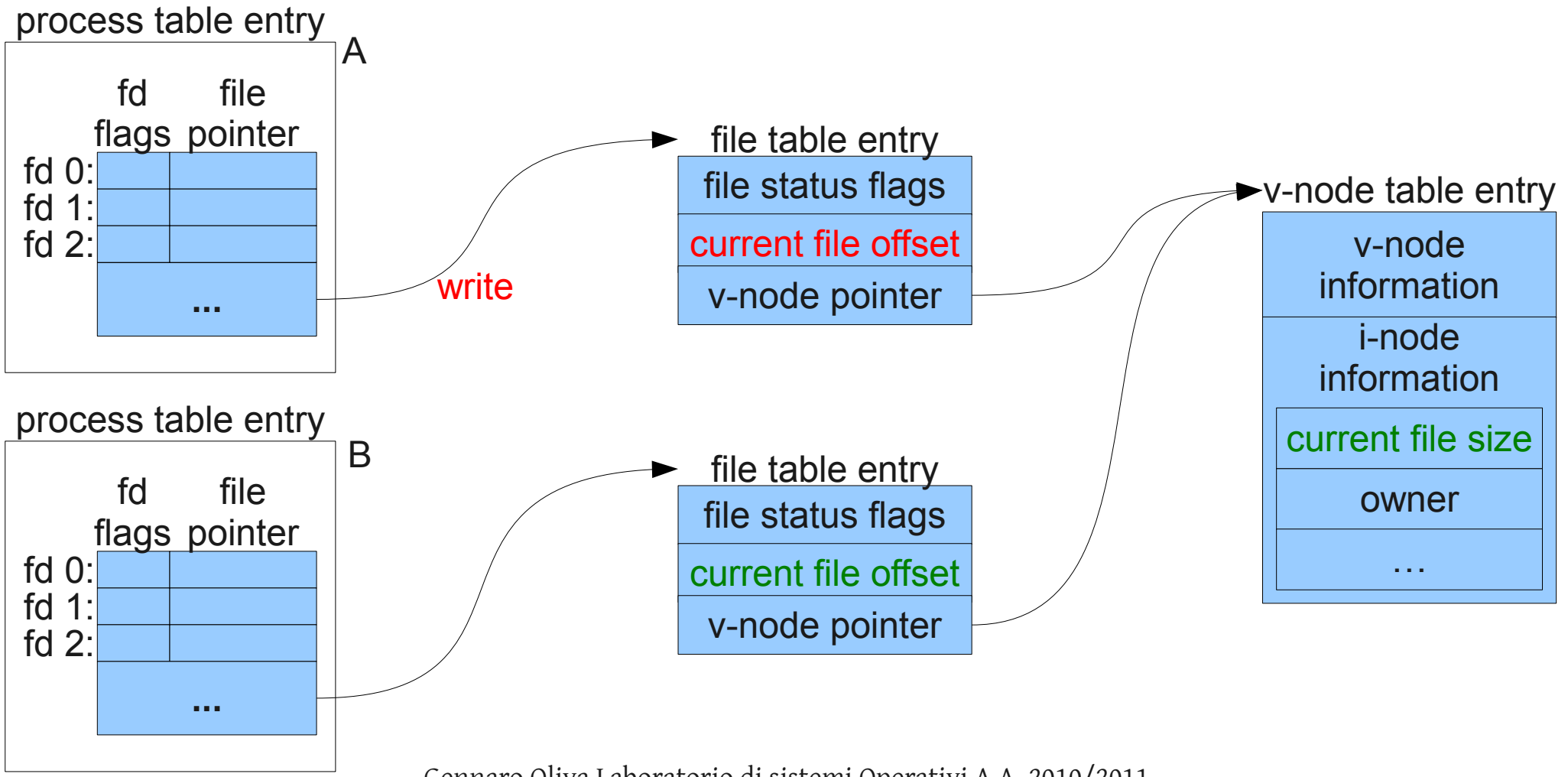
Processi concorrenti e file comuni

- Si consideri ora che, diversamente da A, il processo B riesca a scrivere in coda al file senza essere interrotto, aggiornando il proprio offset e che il kernel modifichi la dimensione del file



Processi concorrenti e file comuni

- A questo punto quando il processo A ritorna in esecuzione eseguirà la write a partire dal vecchio offset sovrascrivendo l'area scritta dal processo B



O_APPEND

- Il problema è che l'operazione di posizionarsi alla fine del file e di scrivere in coda richiede due system call distinte con il pericolo che il kernel sospenda l'esecuzione del processo tra la prima e la seconda chiamata
- La soluzione sta nel rendere atomica questa operazione atomica (ovvero indivisibile) rispetto agli altri processi
- Il kernel fornisce l'opzione O_APPEND che evita la chiamata ad lseek e assicura che prima di ogni operazione di scrittura l'offset del file venga impostato alla reale dimensione del file, rileggendolo dalla tabella dei v-node

Creazione di un file non esistente

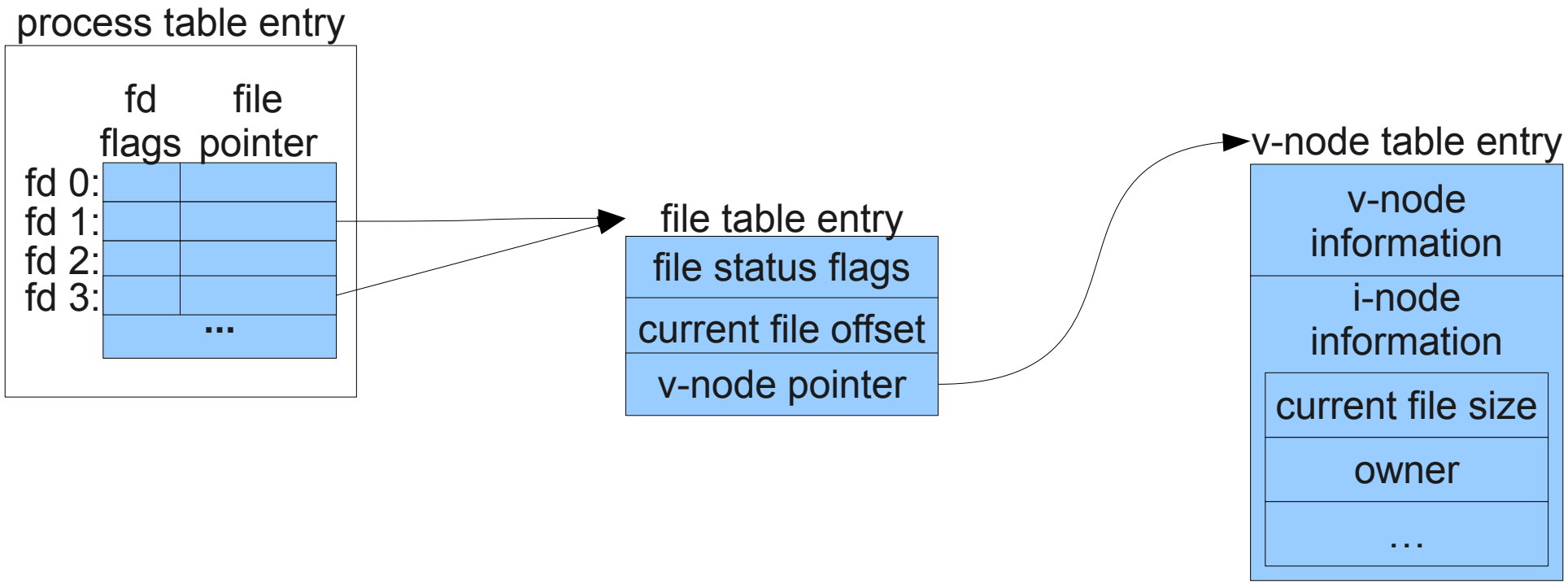
- Una problematica analoga è quella della creazione di un file soltanto se questi non esiste
- Nelle prime versioni di UNIX non era possibile aprire un file che non esiste con la funzione open ma si utilizzava la system call create per la creazione del file e successivamente la open per l'apertura
- Anche in questo caso la necessità di utilizzare due system call può generare problemi se il processo viene interrotto tra la prima e la seconda chiamata
- Per rendere atomica l'operazione, furono introdotti i flag O_CREAT e O_EXCL

Operazioni atomiche

- In generale il termine operazione atomica si riferisce ad un'operazione che si compone di più fasi
- Esistono soltanto due alternative per l'esito dell'esecuzione di un'operazione atomica
  - tutte le fasi sono portate a compimento
  - nessuna fase viene eseguita

Duplicazione di un file descriptor

- Un file descriptor può essere “duplicato” utilizzando le system call dup e dup2
- Dopo la duplicazione, le voci della process table relative ai due file descriptor (quello originale e la copia) puntano allo stesso elemento della file table



dup

- SYNOPSIS

```
#include <unistd.h>

int dup(int srcfd);
```

- DESCRIPTION
  - La system call dup duplica il file descriptor srcfd
- RETURN VALUE
  - La system call restituisce il file descriptor duplicato utilizzando il minimo intero ancora non associato a nessun file
  - In caso di errore dup restituisce -1

dup2

- SYNOPSIS

```
#include <unistd.h>

int dup2(int srcfd, int destfd);
```

- DESCRIPTION
  - La system call dup2 La system call dup duplica il file descriptor srcfd e lo associa al file descriptor destfd
  - Se destfd è aperto, dup2 chiude il file ad esso associato prima di duplicare il descrittore srcfd
  - In tal caso il file puntato da destfd viene chiuso, dopodichè la voce destfd della process table viene indirizzata all'entry della file table a cui punta srcfd
  - La dup2 potrebbe essere simulata attraverso una sequenza di open, close e dup ma l'utilizzo della system call dup2 assicura l'atomicità della sequenza
- RETURN VALUE
  - La system call restituisce il file descriptor duplicato
  - In caso di errore dup2 restituisce -1