

I/O non bufferizzato

Capitolo 3 -- Stevens

System Call

- open
- creat
- umask
- close
- read
- write
- lseek
- dup e dup2

file descriptor

- sono degli interi non negativi
- il kernel assegna un file descriptor ad ogni file aperto
- le funzioni di I/O identificano i file per mezzo dei fd
 - nota la differenza con ANSI C
 - fopen, fclose → FILE *file_pointer

file descriptor...ancora

- per riferirsi ai file si comunica con il kernel tramite i file descriptor
- all'apertura/creazione di un file, il kernel restituisce un **fd** al processo
- da questo momento per ogni operazione su file usiamo il **fd** per riferirci ad esso
- $0 \leq \text{fd} < \text{OPEN_MAX}$
 - ...limiti di OPEN_MAX
 - 19 in vecchie versioni di UNIX
 - 63 in versioni più recenti
 - Limitato dalla quantità di memoria nel sistema (SVR4)

standard file

- Ogni nuovo processo apre 3 file standard
 - input
 - output
 - error
- e vi si riferisce con i tre file descriptor
 - **0** (STDIN_FILENO)
 - **1** (STDOUT_FILENO)
 - **2** (STDERR_FILENO)

open

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

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

Restituisce: un **fd** se OK
-1 altrimenti

- **fd** restituito è il più piccolo numero non usato come **fd**

open

- L'argomento *oflag* è formato dall'OR di uno o più dei seguenti **flag di stato**
 - **Una ed una sola costante tra**
 - O_RDONLY, O_WRONLY, O_RDWR
 - **Una qualunque tra (sono opzionali)**
 - O_APPEND = tutto ciò che verrà scritto sarà posto alla fine
 - O_CREAT = usato quando si usa open per creare un file
 - O_EXCL = messo in Or con O_CREAT per segnalare errore se il file già esiste
 - O_TRUNC = se il file già esiste, aperto in write op read-write tronca la sua lunghezza a 0
 - O_SYNC (SVR4) = se si sta aprendo in write, fa completare prima I/O
 - O_NOCTTY, O_NONBLOCK

open

- L'argomento ***mode*** viene utilizzato quando si crea un nuovo file utilizzando **O_CREAT** per specificare i permessi di accesso del nuovo file che si sta creando. Se il file già esiste questo argomento viene ignorato.

Costanti per il mode

<i>mode</i>	Description	
S_ISUID	set-user-ID on execution	4000
S_ISGID	set-group-ID on execution	2000
S_ISVTX	saved-text (sticky bit)	1000
S_IRWXU	read, write, and execute by user (owner)	0700
S_IRUSR	read by user (owner)	0400
S_IWUSR	write by user (owner)	0200
S_IXUSR	execute by user (owner)	0100
S_IRWXG	read, write, and execute by group	0070
S_IRGRP	read by group	0040
S_IWGRP	write by group	0020
S_IXGRP	execute by group	0010
S_IRWXO	read, write, and execute by other (world)	0007
S_IROTH	read by other (world)	0004
S_IWOTH	write by other (world)	0002
S_IXOTH	execute by other (world)	0001

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

int main(void)
{
    int    fd;

    fd=open("FILE",O_RDONLY);
    exit(0);
}
```

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

int main(void)
{
    int    fd;
    fd=open("FILE1",O_CREAT|O_EXCL|O_WRONLY,0600);
    exit(0);
}
```

creat

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

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

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

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

Descrizione: crea un file dal nome *pathname* con i permessi descritti in *mode*

Restituisce: fd del file aperto come write-only se OK,
-1 altrimenti

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

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

umask

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

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

```
mode_t umask (mode_t cmask);
```

Descrizione: imposta la maschera di creazione per l'accesso ad un file

Restituisce: la maschera di creazione precedente (nota che non restituisce valori di errore)

umask

- Viene usato ogni volta che il processo crea un nuovo file o directory, secondo il seguente criterio:
 - setta la maschera di creazione (*cmask*)
 - comando: **umask**
 - alla creazione del file viene fatto l'AND tra la maschera negata e il *mode* della creazione del file

umask: 022 (--- -w- -w-)	000 010 010	NOT
---------------------------	-------------	-----

negaz.: 755 (rwx r-x r-x)	111 101 101	AND
---------------------------	-------------	-----

creat("pippo.txt", 0665)	<u>110 110 101</u>
--------------------------	--------------------

rw- r- r-x	110 100 101
------------	-------------



- qual'è la maschera di default di creazione di file?

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

        /* esempio di utilizzo di umask */

int main(void)
{
    umask(0);
    if(creat("foo",S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)<0)
        printf("creat error for foo \n");

    umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);    /* 0066 */

    if (creat("bar",S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)<0)
        printf("creat error for bar \n");
    exit(0);
}
```

Risultati programma precedente

- rw- rw- rw- foo
- rw- --- --- bar

close

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

Descrizione: chiude il file con file descriptor *filedes*

Restituisce: 0 se OK
 -1 altrimenti

Quando un processo termina, tutti i file aperti vengono automaticamente chiusi dal kernel

offset

- ogni file aperto ha assegnato un **current offset** (intero >0) che misura in numero di byte la posizione nel file
- Operazioni come **open** e **creat** settano il current offset all'inizio del file a meno che `O_APPEND` sia specificato (open)
- operazioni come **read** e **write** partono dal current offset e causano un incremento pari al numero di byte letti o scritti

`lseek`

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

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

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

Restituisce: il nuovo offset se OK

-1 altrimenti

lseek

L'argomento *whence* può assumere valore

- **SEEK_SET**

- ci si sposta del valore di *offset* a partire dall'inizio

- **SEEK_CUR**

- ci si sposta del valore di *offset* (positivo o negativo) a partire dalla posizione corrente

- **SEEK_END**

- ci si sposta del valore di *offset* (positivo o negativo) a partire dalla fine (taglia) del file

Iseek

- Iseek permette di settare il current offset oltre la fine dei dati esistenti nel file
- Se vengono inseriti successivamente dei dati in tale posizione, si crea un **buco**
- Una lettura nel buco restituirà byte con valore 0 (**\0**)
- In ogni caso Iseek non aumenta la taglia del file
- Se Iseek fallisce (restituisce -1), il valore del current offset rimane inalterato

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

int main(void)
{
    int    fd,i;

    fd=open( "FILE", O_RDONLY);
    i=lseek( fd, 50, SEEK_CUR);
    exit(0);
}
```

read

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

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

Descrizione: legge dal file con file descriptor *filedes* un numero di byte *nbyte* e li mette in *buff*

Restituisce: il numero di bytes letti,

- 0 se alla fine del file

- 1 altrimenti

read

- La lettura parte dal current offset
- Alla fine il current offset è incrementato del numero di byte letti
- Se *nbytes*=0 viene restituito 0 e non vi è altro effetto
- Se il current offset è alla fine del file o anche dopo, viene restituito 0 e non vi è alcuna lettura


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

int main(void)
{
    int    fd, i;
    char    buf[20];

    fd=open("FILE", O_RDONLY);
    i=lseek(fd, 50, SEEK_CUR);
    read(fd, buf, 20);
    exit(0);
}
```

write

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

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

Descrizione: scrive *nbyte* presi dal *buff* sul file con file descriptor *filedes*

Restituisce: il numero di byte scritti se OK
-1 altrimenti

write

- La posizione da cui si comincia a scrivere è current offset
- Alla fine della scrittura current offset è incrementato di *nbytes* e se tale scrittura ha causato un aumento della lunghezza del file anche tale parametro viene aggiornato
- Se viene richiesto di scrivere più byte rispetto allo spazio a disposizione (es: limite fisico di un dispositivo di output), solo lo spazio disponibile è occupato e viene restituito il numero effettivo di byte scritti ($\leq \textit{nbytes}$)
- Se *filedes* è stato aperto con O_APPEND allora current offset è settato alla fine del file in ogni operazione di write
- Se *nbytes*=0 viene restituito 0 e non vi è alcuna scrittura

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

int main(void)
{
    int    fd,i,n;
    char    buf[20];

    fd=open("FILE",O_RDONLY);
    i=lseek(fd,50,SEEK_CUR);
    n=read(fd,buf,20);
    write(1,buf,n);
    exit(0);
}
```

```
/* File: seek.c */
#include <sys/types.h>
#include <fcntl.h>

int main(void)
{
    off_t    i;
    int      fd;
    char     *s;

    fd=open("seek.c", O_RDONLY);
    i=lseek(fd, 30, SEEK_CUR);
    printf("posizione corrente  %d\n", i);

    s=(char *) malloc(25*sizeof(char));
    read (fd, s, 20);
    printf ("leggo da: \n %s\n", s);
    exit(0);
}
```

```
/* File: seek.c */
#include <sys/types.h>
#include <fcntl.h>

int main(void)
{
    off_t    i;
    int      fd;
    char     s[25];

    fd=open("seek.c", O_RDONLY);
    i=lseek(fd, 30, SEEK_CUR);
    printf("posizione corrente   %d\n", i);

    read (fd, s, 20);
    printf ("leggo da: \n %s\n", s);
    exit(0);
}
```

Efficienza di I/O

```
#define BUFFSIZE 8192

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

    while((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            printf("write error");
        if (n < 0)    printf("read error");
        exit(0);
}
```

-apertura file standard
-chiusura file

- scelta di BUFFERSIZE

Efficienza di I/O

BUFFSIZE	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	Number of loops
1	20.03	117.50	138.73	516,581,760
2	9.69	58.76	68.60	258,290,880
4	4.60	36.47	41.27	129,145,440
8	2.47	15.44	18.38	64,572,720
16	1.07	7.93	9.38	32,286,360
32	0.56	4.51	8.82	16,143,180
64	0.34	2.72	8.66	8,071,590
128	0.34	1.84	8.69	4,035,795
256	0.15	1.30	8.69	2,017,898
512	0.09	0.95	8.63	1,008,949
1,024	0.02	0.78	8.58	504,475
2,048	0.04	0.66	8.68	252,238
4,096	0.03	0.58	8.62	126,119
8,192	0.00	0.54	8.52	63,060
16,384	0.01	0.56	8.69	31,530
32,768	0.00	0.56	8.51	15,765
65,536	0.01	0.56	9.12	7,883
131,072	0.00	0.58	9.08	3,942
262,144	0.00	0.60	8.70	1,971
524,288	0.01	0.58	8.58	986

Figure 3.6 Timing results for reading with different buffer sizes on Linux

Esercizio 2.1

- Programma 3.2 del libro di testo: buco
 - Creare un file FILE1 contenente un buco
 - far stampare il contenuto del file col buco
 - od -c
 - cat
 - Copiare il contenuto di FILE1 in un file FILE2 eliminando il buco

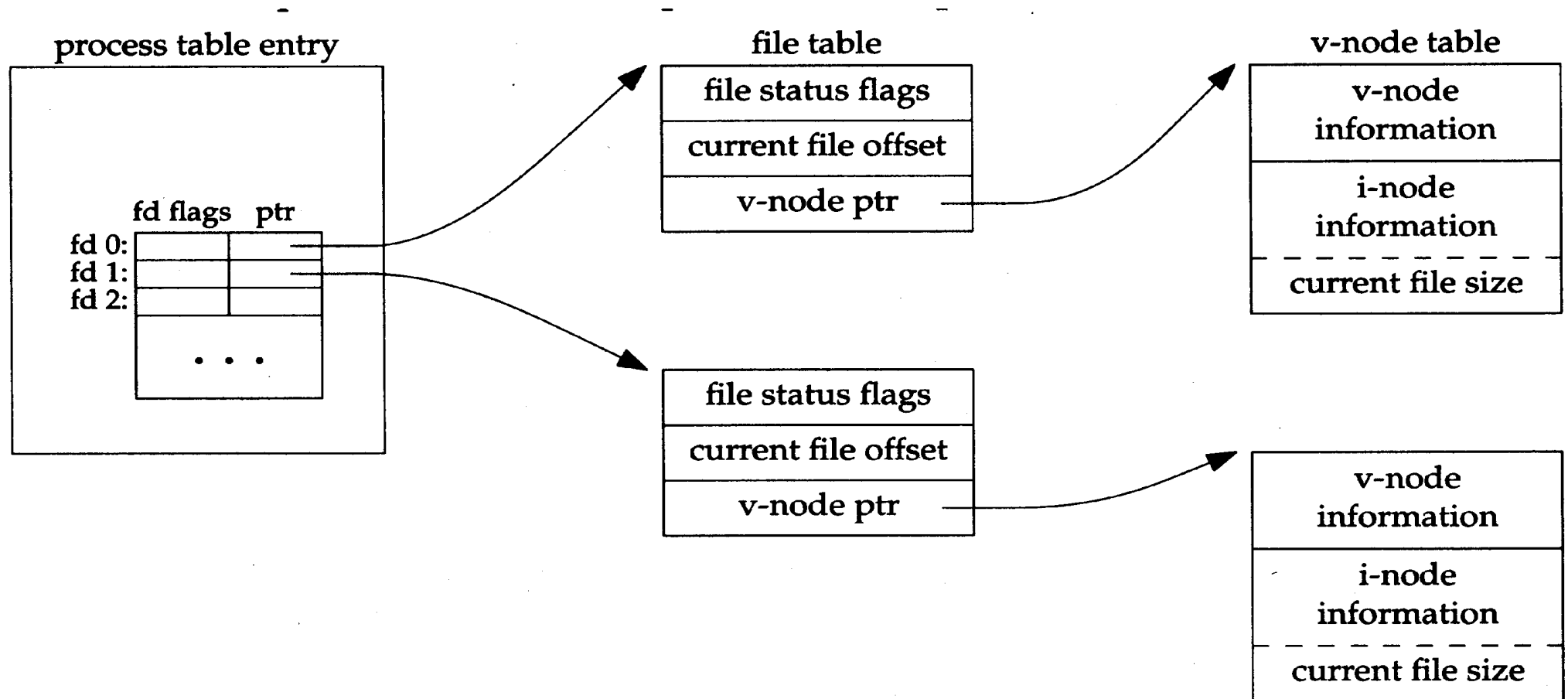
Esercizio 2.2

- Scrivere un programma in C che prende un input da tastiera e lo scrive nel file FILE1.
- Copiare in ordine inverso il contenuto di FILE1 in un file FILE2 e stampare il contenuto di FILE2 sul terminale.

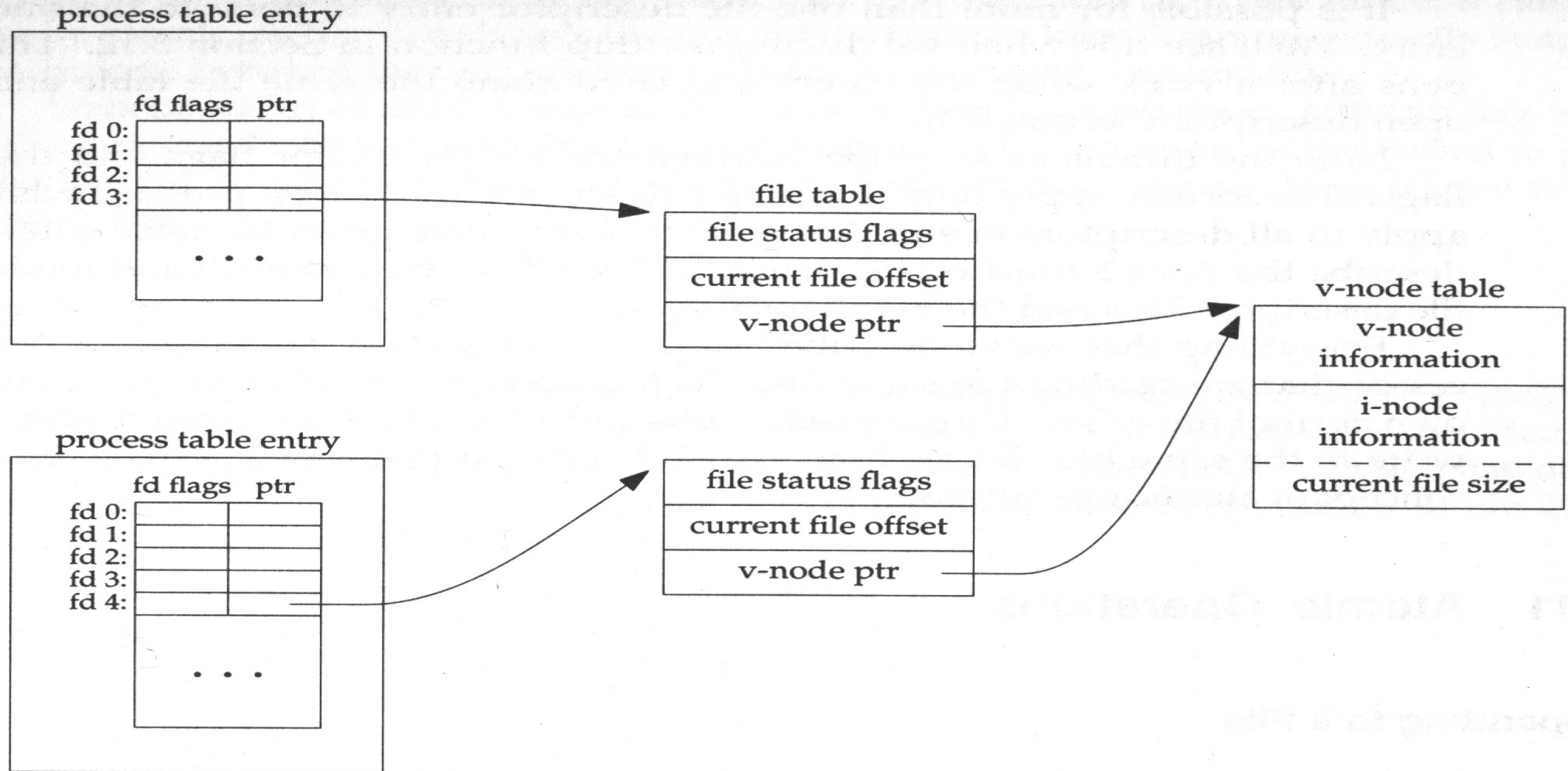
Condivisione di file

- Unix supporta la possibilità che più processi condividano file aperti
- Prima di analizzare questa situazione esaminiamo le strutture dati che il kernel utilizza per I/O
 - 3 strutture dati per l'I/O

Strutture dati di file aperti



2 processi su uno stesso file




Operazioni Atomiche

- Immaginate il seguente scenario:
 - 2 processi aprono lo stesso file
 - ognuno si posiziona alla fine e scrive (in 2 passi)
 1. `lseek(fd, 0 ,SEEK_END);`
 2. `write (fd, buff , 100);`
 - se il kernel alterna le due operazioni di ogni processo si hanno

effetti indesiderati

Operazioni Atomiche

- Unix risolve il problema:
 - Apre il file con il flag "O_APPEND"
 - Questo fa posizionare l'offset alla fine, prima di ogni write
 - In altre parole, le operazioni di
 1. posizionamento
 2. write sono atomiche

In generale una **operazione atomica** è composta da molti passi che o sono eseguiti tutti insieme o non ne è eseguito nessuno

dup & dup2

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

```
int dup( int filedes );
```

```
int dup2( int filedes, int filedes2 );
```

Descrizione: assegnano un altro fd ad un file che già ne possedeva uno, cioè *filedes*

Restituiscono entrambe: il nuovo fd se OK

-1 altrimenti

dup & dup2

- In particolare:

```
int dup( int filedes );
```

- restituisce il più piccolo fd disponibile

```
int dup2( int filedes, int filedes2 );
```

- ▶ Assegna al file avente già file descriptor *filedes* anche il file descriptor *filedes2*
 - Se *filedes2* è già open esso è prima chiuso e poi è assegnato a *filedes*
 - Se *filedes2*=*filedes* viene restituito direttamente *filedes2*
- ▶ dup2 è una operazione atomica

Situazione dopo dup

Notare che il nuovo fd
condivide con il vecchio la
stessa entry nella file table

process table entry

	fd flags	ptr
fd 0:		
fd 1:		
fd 2:		
fd 3:		
	...	

file table

file status flags
current file offset
v-node ptr

v-node table

v-node information
i-node information

current file size

Esercizi 2.3 e 2.4

1. copiare un file in un altro usando solo le funzioni di standard I/O *getchar* e *putchar*
 - hint: "duplicare" gli standard file
2. copiare il contenuto di un file in un altro usando esclusivamente **read** da standard input e **write** su standard output