Chapter 3. File I/O

朱金辉

华南理工大学软件学院

1. Introduction

- Discussion of unbuffered I/O
- 5 functions:
  - open()
  - read()
  - write()
  - lseek()
  - close()
- Sharing of files: dup(), fcntl(), ioctl()

2. File Descriptors

- A nonnegative integer
- Represents a file
- Referenced by the kernel
- Returned by open() to process
- read(), write() use them
- stdin=0, stdout=1, stderr=2

3. open()

- #include <sys/types.h>
- #include <sys/stat.h>
- #include <fcntl.h>
- int open(const char **pathname*, int *oflag*, …/* , mode_t *mode* */);
- Returns: file descriptor if OK, -1 on error

open()

- *pathname*: name of file to open
- *oflag* = one of the following:
  - O_RDONLY: open for reading only
  - O_WRONLY: open for writing only
  - O_RDWR: open for reading & writing
- Only 1 of the above must be specified

open()

- Other optional constants:
  - O_APPEND: append to end of file
  - O_CREAT: create if not existing, need mode
  - O_EXCL: gen error if file exist and O_CREAT
  - O_TRUNC: truncate length to 0, if exists
  - O_NOCTTY: do not allocate as controlling tty
  - O_NONBLOCK: nonblocking mode for FIFO or character special file
  - O_SYNC: wait phy I/O to complete (on write)

4. creat()

- #include <sys/types.h>
- #include <sys/stat.h>
- #include <fcntl.h>
- int creat(const char **pathname*,
  mode_t *mode*);
- Returns: file descriptor opened for write-only if OK, -1 on error

creat()

- Equivalent to:
  open(*pathname*, O_WRONLY | O_CREAT | O_TRUNC, *mode*);

5. close()

- #include <unistd.h>
- int close(int *filedes*);
- Returns: 0 if OK, -1 on error
- Releases record locks on the file
- All open files automatically closed by kernel, when a process terminates.
- Take advantage: no explicit close()

6. lseek()

Current file offset:

- nonnegative integer
- #bytes from beginning of file
- read(), write() start from current file offset
- initialized to 0 when file is opened (unless O_APPEND is specified)

lseek()

- #include <sys/types.h>
- #include <unistd.h>
- off_t lseek(int *filedes*, off_t *offset*, int *whence*);
- Returns: new file offset if OK, -1 on error

lseek()

whence	**New offset (bytes)**
SEEK_SET	file start + *offset*
SEEK_CUR	current offset + *offset*
SEEK_END	file size + *offset*

lseek()

- To determine current offset:
  - offt_t curpos;
  - currpos = lseek(fd, 0, SEEK_CUR);
- If fd is a FIFO or pipe:
  - currpos = -1
  - errno = EPIPE

Program 3.1: seeking stdin

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

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

Program 3.1 (output)

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

lseek()

- For regular files, offset is usually nonnegative
- For special files, offset can be negative,

  Check -1 and not test < 0
- lseek() only records current file offset in kernel, no I/O takes place
- offset > file size? OK! Hole in file! All 0!

Program 3.2: Create file with hole

```
#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");
```

Program 3.2: Create file with hole

```
    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);
}
```

Program 3.2 (output)

- $ a.out
- ls –l file.hole

-rw-r--r– 1 stevens 50 Jul 31 0:50 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 \0
0000040  \0 \0 \0 \0 \0 \0 \0 \0 A B C D E F G H
0000060  I J
```

(–c: print as characters)

7. read()

- #include <unistd.h>
- ssize_t read(int *filedes*, void **buff*,
  size_t *nbytes*);
- Returns: #bytes read, 0 if EOF, -1 on error
- #bytes read may be < nbytes

read()

When is #bytes read < nbytes requested?

- EOF reached
- Terminal: 1 line at a time
- Network: buffer size limit
- Magnetic tape: single record at a time

8. write()

- #include <unistd.h>
- ssize_t write(int filedes, const void *buff, size_t nbytes);
- Returns: #bytes written if OK, -1 on error
- File offset incremented by #bytes written

Program 3.5: stdout ← stdin

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

Program 3.4 (details)

- stdin and stdout opened automatically by UNIX shell
- STDIN_FILENO (assumed as 0)
- STDOUT_FILENO (assumed as 1)
- stdin and stdout are not closed
- No difference between text and binary files

9. I/O Efficiency (Buffer size?)

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

No effect on increasing beyond 4096 bytes

mytest

```
→  fileio git:(master) ✗ time ./test_fileio 1 <~/test.data >/dev/null
./test_fileio 1 < ~/test.data > /dev/null  18.84s user 139.50s system 98% cpu 2:40.01 total
→  fileio git:(master) ✗ time ./test_fileio 1024 <~/test.data >/dev/null
./test_fileio 1024 < ~/test.data > /dev/null  0.02s user 0.21s system 84% cpu 0.275 total
→  fileio git:(master) ✗ time ./test_fileio 4096 <~/test.data >/dev/null
./test_fileio 4096 < ~/test.data > /dev/null  0.00s user 0.11s system 70% cpu 0.158 total
→  fileio git:(master) ✗ time ./test_fileio 524288 <~/test.data >/dev/null
./test_fileio 524288 < ~/test.data > /dev/null  0.00s user 0.08s system 60% cpu 0.139 total
```

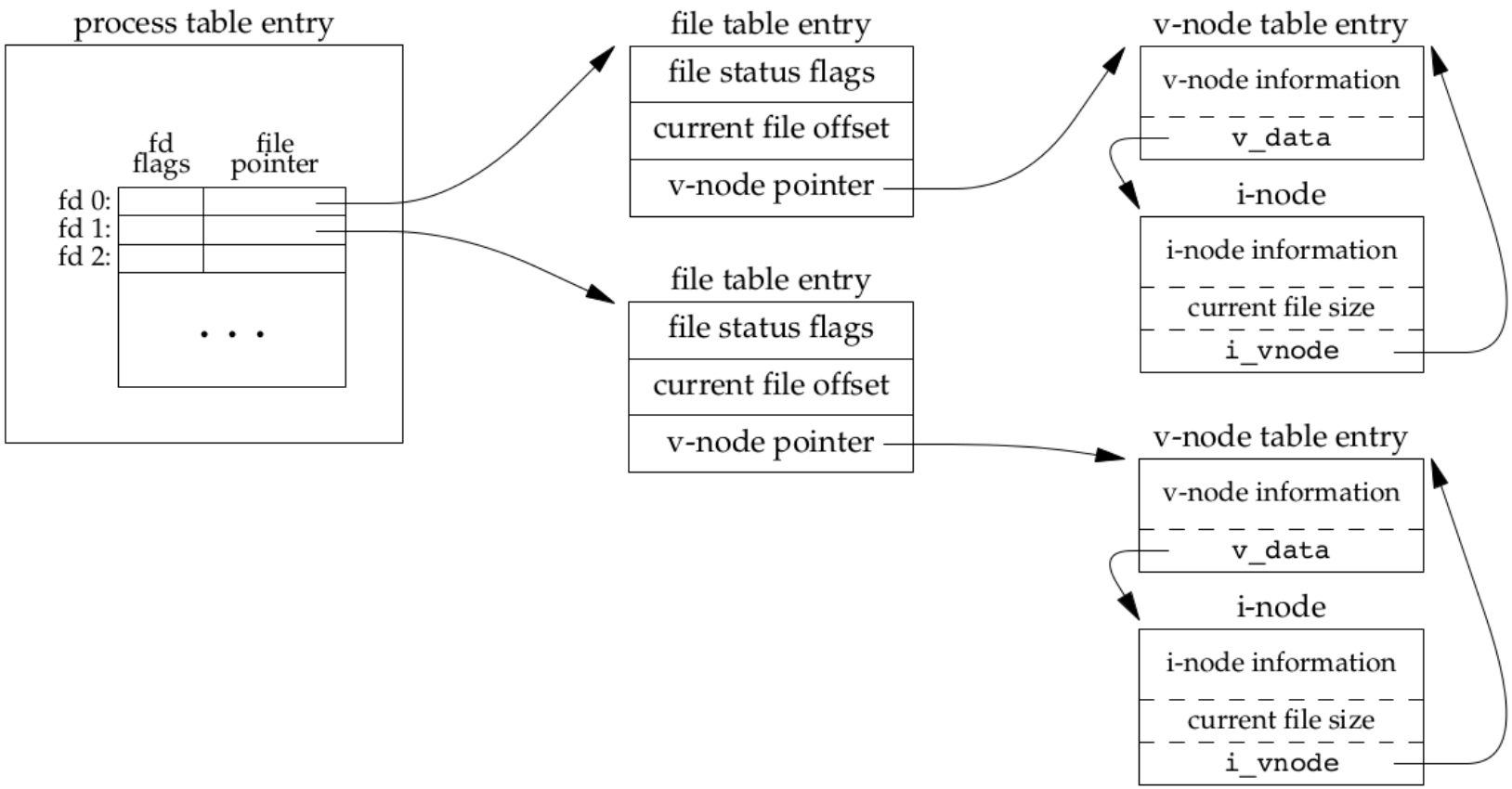
10. File Sharing

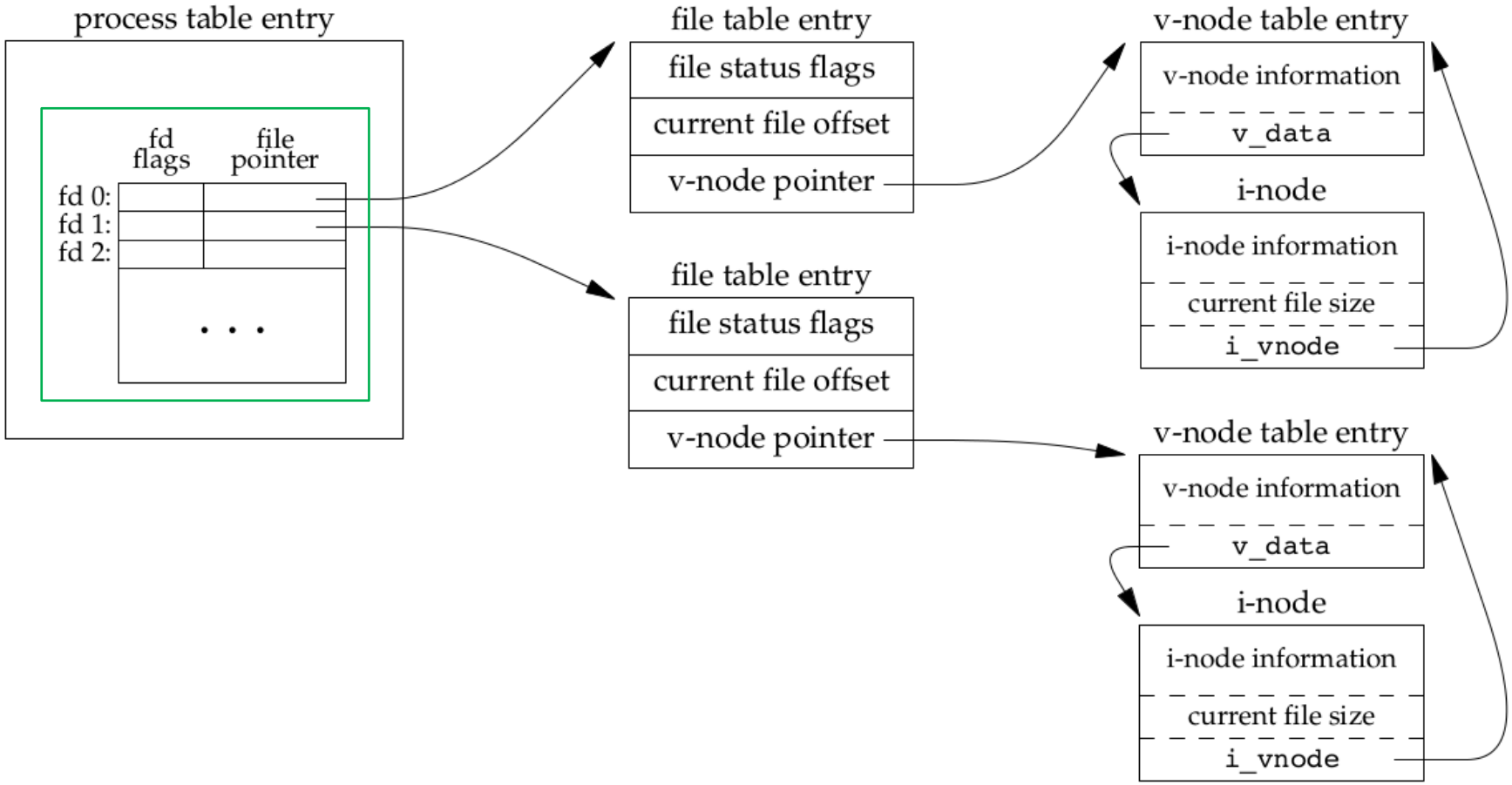
Unix supports the sharing of open files between different processes.

Three data structures used by kernel

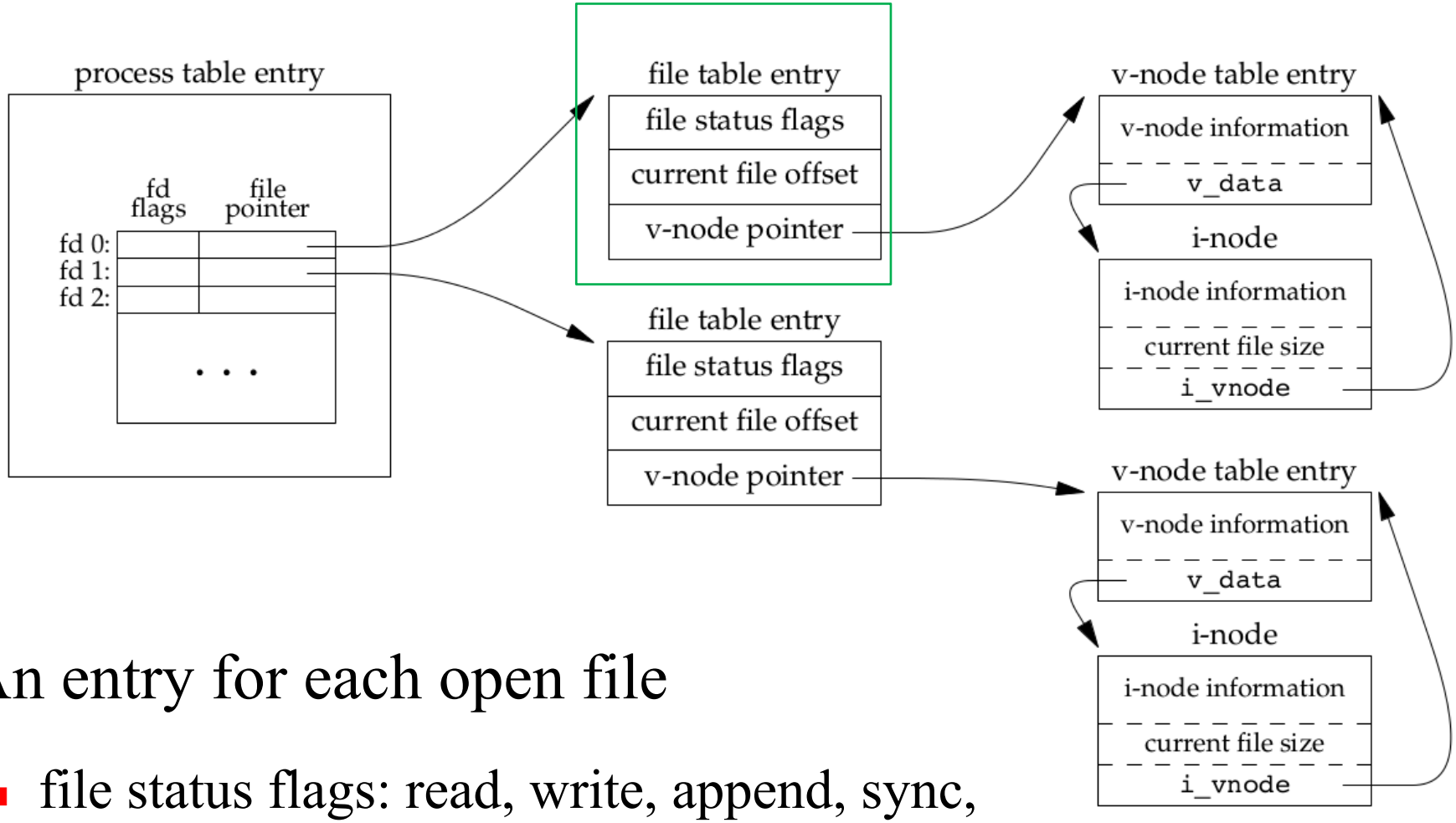
- Process Table
- File Table
- V-node Table

Kernel data structures for open files

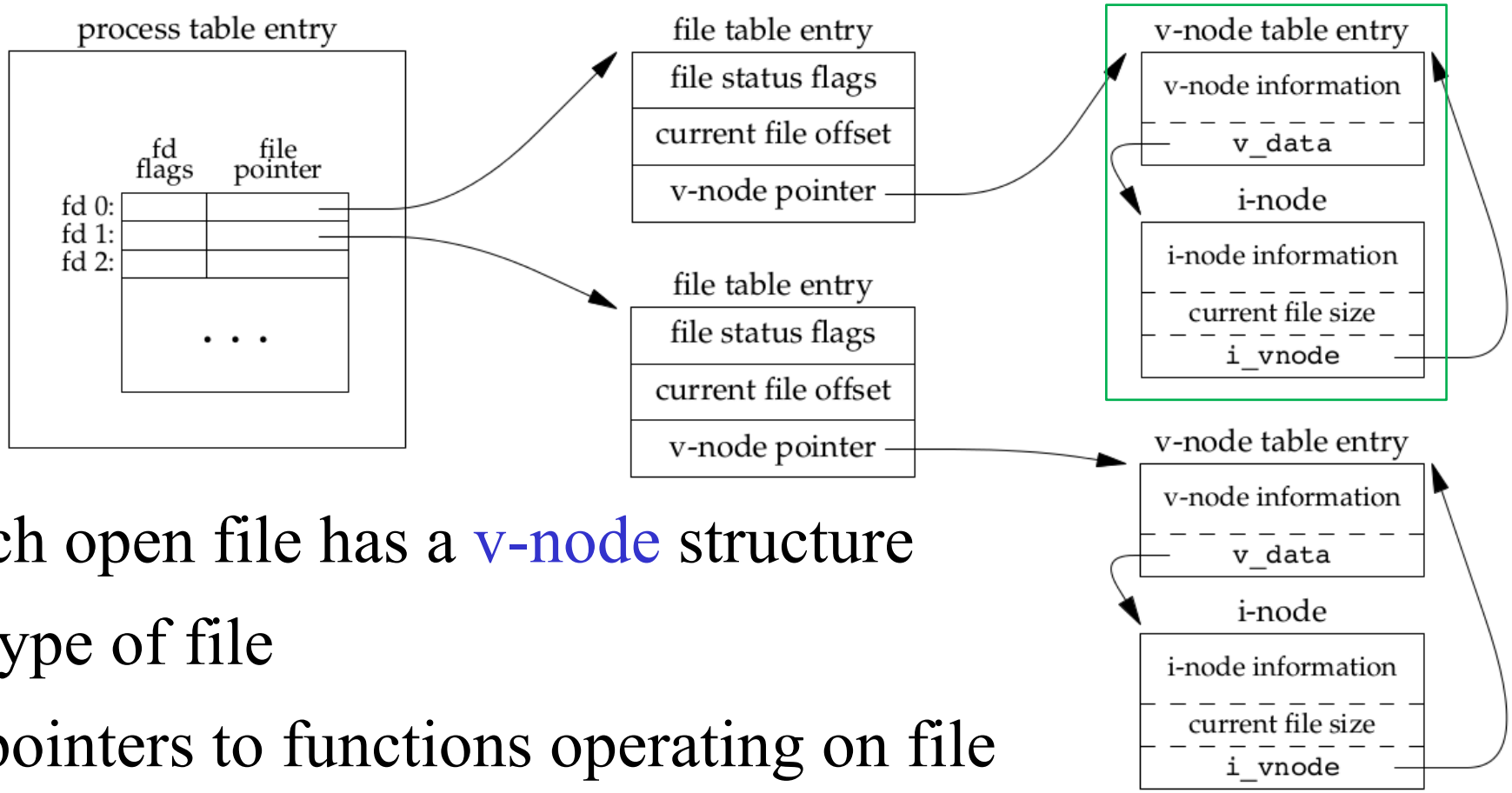




- A table of open file descriptors in each entry
  - File descriptor flag
  - A pointer to a file table entry



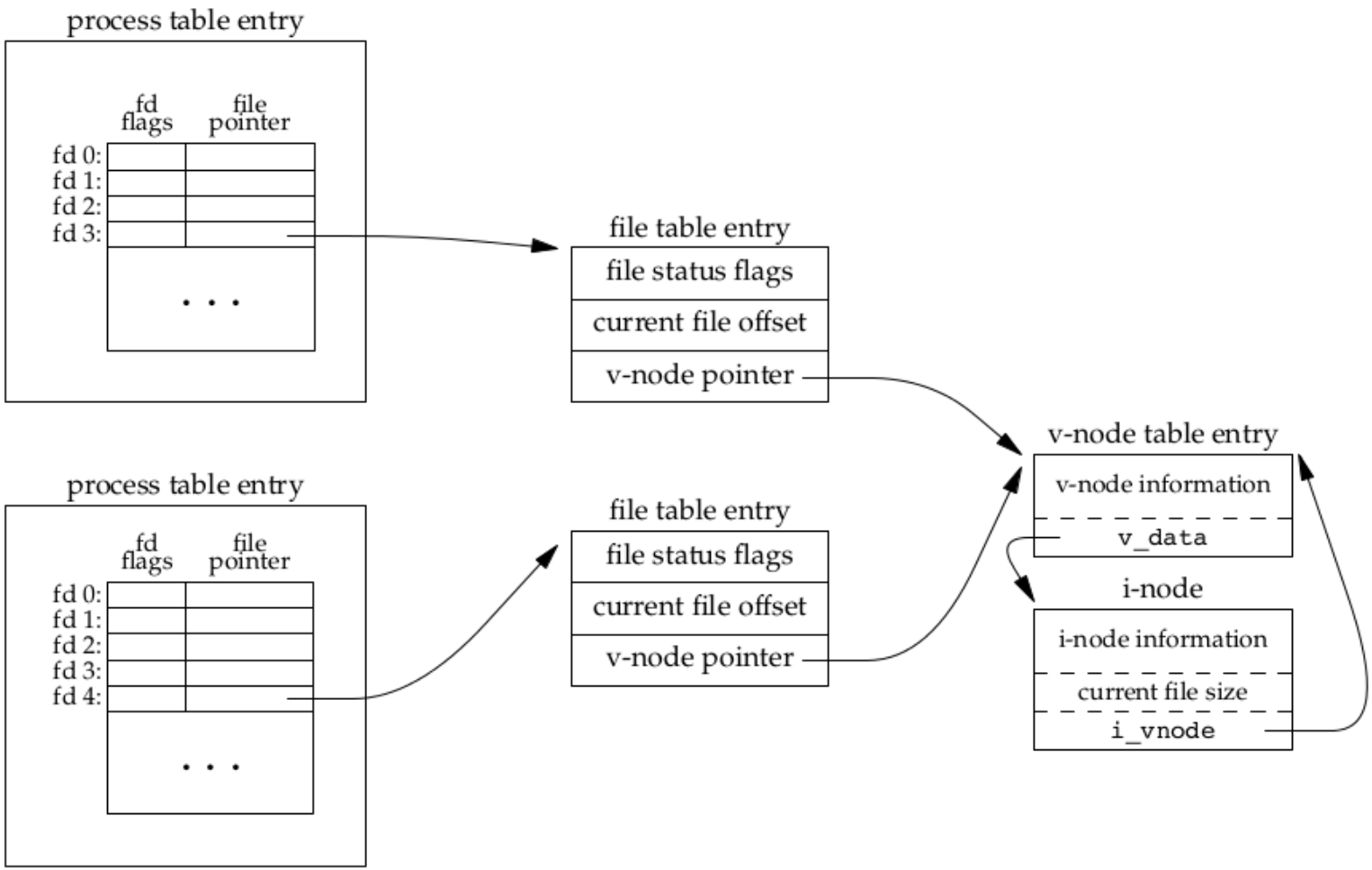
- An entry for each open file
  - file status flags: read, write, append, sync, nonblocking, etc.
  - current file offset
  - a pointer to a v-node table entry for the file



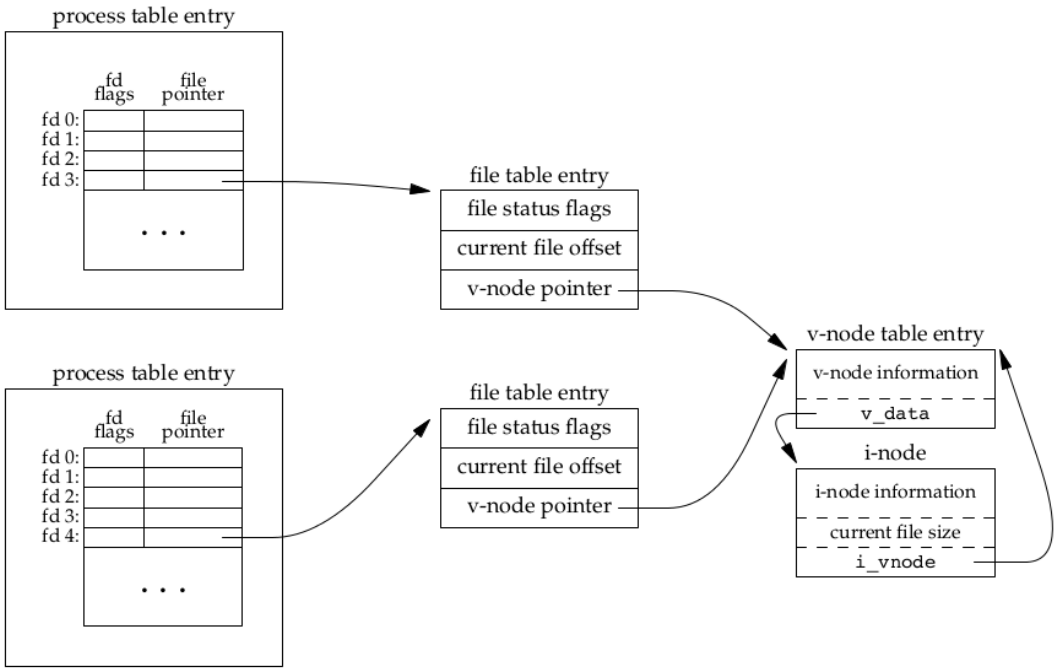
Each open file has a v-node structure

- type of file
- pointers to functions operating on file
- i-node for the file:
  - owner,
  - size,
  - device,
  - disk location pointers

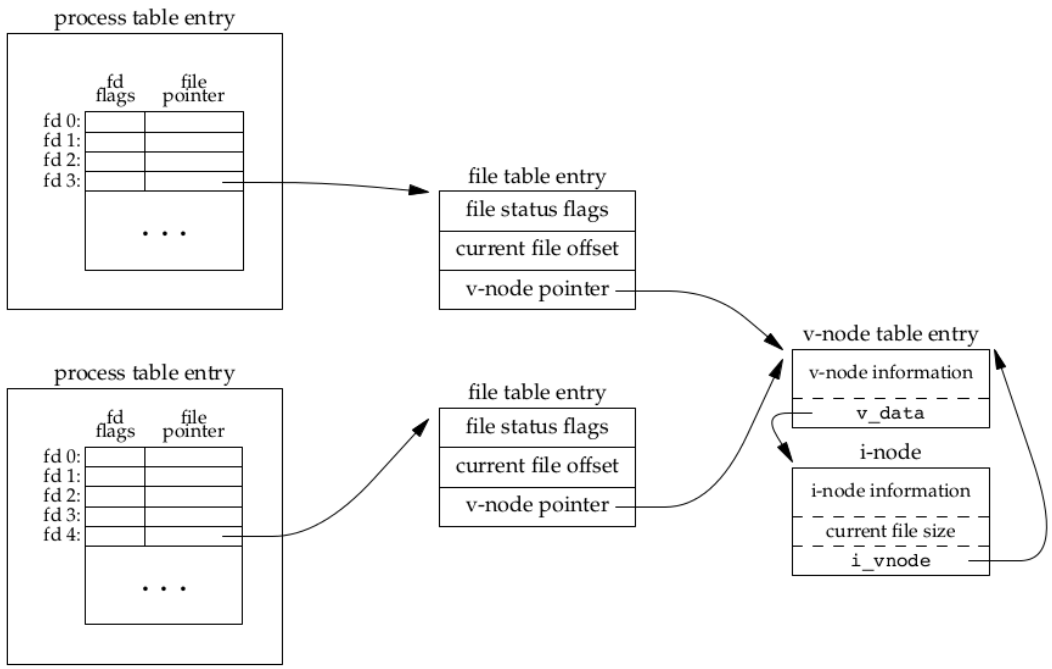
File Sharing



- After each write(), current file offset is incremented by #bytes written.
- If file size increases, current file size in i-node table is updated.
- If a file open() has O_APPEND flag, file status flags are updated and before each write, current file offset := current file size (from i-node table)



- lseek() only modifies current file offset. No I/O.
- lseek(fd, 0, SEEK_END);
  current file offset := current file size
- All above work fine for multiple processes reading same file.
- For write() of same file by multiple processes, unexpected results can occur



11. Atomic Operations

Consider the following program:

```
if(lseek(fd, 0L, 2)<0)  /*position to EOF*/
        err_sys("lseek error");
if(write(fd, buff, 100) != 100)
        err_sys("write error");
```

?

Atomic Operations

Scenario

Processes A & B appending to same file

- Set file offset to EOF (1500) for A
- Kernel switches to Process B
- Set file offset to EOF (1500) for B
- Write by B, file size increased to 1600
- Kernel switches back to Process A
- Write by A, data of B is **overwritten**!!!

Atomic Operations (APPEND)

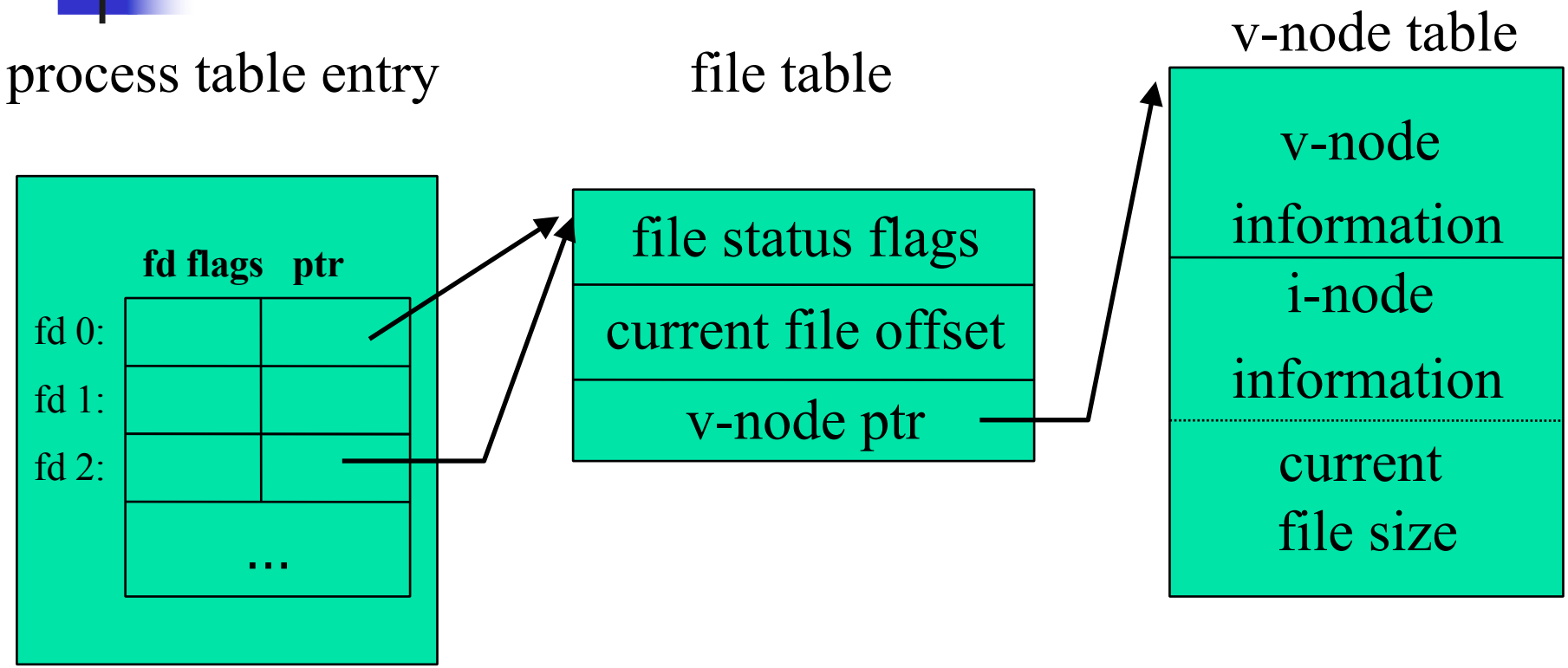
- What is the problem?
- Append is implemented as 2 actions:
  - position to EOF
  - write to file
- Kernel may switch from one process to another **between those two actions**
- Solution: let append be **ATOMIC**!!!

O_APPEND

12. dup() and dup2()

- #include <unistd.h>
- int dup (int *filedes*);
- int dup2 (int *filedes*, int *filedes2*);
- Return: new file descriptor if OK, -1 on error
- Duplicate *filedes* and return as new file descriptor,
  - dup: lowest numbered file descriptor
  - dup2: specified as *filedes2*

Kernel data structures on dup(1)



13. sync, fsync, and fdatasync

- Delayed Write: When we write data to a file, the data is normally copied by the kernel into one of its buffers and queued for writing to disk at some later time.

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

- int fsync(int fd);
- int fdatasync(int fd);
- void sync(void);

Returns: 0 if OK, −1 on error

sync & fsync

- The sync function simply queues all the modified block buffers for writing and returns; it does not wait for the disk writes to take place.
- The function fsync refers only to a single file, specified by the file descriptor fd, and waits for the disk writes to complete before returning. This function is used when an application, such as a database, needs to be sure that the modified blocks have been written to the disk.
- The fdatasync function is similar to fsync, but it affects only the data portions of a file. With fsync, the file's attributes are also updated synchronously.

14. fcntl()

- #include <sys/types.h>
- #include <unistd.h>
- #include <fcntl.h>
- int fcntl (int *filedes*, int *cmd*, …
  /* int *arg* */);
- Returns: depends on cmd if OK,
  -1 on error

fcntl()

Change properties of an opened file

cmd	fcntl() results
F_DUPFD	duplicate an existing descriptor
F_GETFD or F_SETFD	get/set file descriptor flags
F_GETFL or F_SETFL	get/set file status flags
F_GETOWN or F_SETOWN	get/set async I/O owner
F_GETLK, F_SETLK, or F_SETLKW	get/set record locks

Program 3.11: print file flags

```
#include        <sys/types.h>
#include        <fcntl.h>
#include        "apue.h"

int main(int argc, char *argv[]) {
    int             accmode, val;

    if (argc != 2)
            err_quit("usage: a.out <descriptor#>");

    if ( (val = fcntl(atoi(argv[1]), F_GETFL, 0)) < 0)
            err_sys("fcntl error for fd %d", atoi(argv[1]));
```

Program 3.11: print file flags

```
    accmode = val & O_ACCMODE;
    if      (accmode == O_RDONLY)          printf("read only");
    else if (accmode == O_WRONLY)          printf("write only");
    else if (accmode == O_RDWR) printf("read write");
    else err_dump("unknown access mode");

    if (val & O_APPEND)                    printf(", append");
    if (val & O_NONBLOCK)                  printf(", nonblocking");
#if !defined(_POSIX_SOURCE) && defined(O_SYNC)
    if (val & O_SYNC)                      printf(", synchronous writes");
#endif
    putchar('\n');
    exit(0);
}
```

Program 3.11: (output)

- **$ a.out 0 < /dev/tty**
- read only
- **$ a.out 1 > temp.foo**
- **$ cat temp.foo**
- write only
- **$ a.out 2 2>>temp.foo**
- write only, append
- **$ a.out 5 5<>temp.foo**
- read write

Program 3.12: set flags

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

void
set_fl(int fd, int flags) /* flags are file status flags to turn on */
{
    int                 val;

    if ( (val = fcntl(fd, F_GETFL, 0)) < 0)
            err_sys("fcntl F_GETFL error");

    val |= flags;                   /* turn on flags */

    if (fcntl(fd, F_SETFL, val) < 0)
            err_sys("fcntl F_SETFL error");
}
```

Program 3.12: clear flags

- How to clear flags for a file?
- val &= ~flags; /* turn flags off */

15. ioctl()

- #include <unistd.h> /* SVR4 */
- #include <sys/ioctl.h> /* 4.3+ BSD */
- int ioctl(int *filedes*, int *request*, …);
- Returns: -1 on error, something else if OK
- not POSIX.1, but SVR4 & 4.3+BSD use it for many device operations

ioctl()

Category	Constant names	Header	Number of ioctls
disk labels	DIOxxx	<sys/disklabel.h>	4
file I/O	FIOxxx	<sys/filio.h>	14
mag tape I/O	MTIOxxx	<sys/mtio.h>	11
socket I/O	SIOxxx	<sys/sockio.h>	73
terminal I/O	TIOxxx	<sys/ttycom.h>	43

Figure 3.15 Common FreeBSD ioctl operations

ioctl()

- It can be used on magetic tapes to:
  - write end-of-file marks
  - rewind tape
  - space forward over #files or #records
- No other function (read, write, lseek) can be used for the above

Summary of I/O buffering