File I/O and Standard I/O Library Chapters 3 and 5

Cheng-Hsin Hsu
National Tsing Hua University
Department of Computer Science

Parts of the course materials are courtesy of Prof. Chun-Ying Huang

Outline

- Introduction
- File I/O Functions
- File I/O Issues
- Standard I/O Functions

Introduction

Standard Input, Output, and Error

- A shell creates standard input, standard output, and standard error when it runs a program
- Standard input, output, and error can be *piped* and/or *redirected*
- Examples – The "`cat`" program
  - `$ cat /etc/passwd`
  - `$ cat < /etc/passwd`
  - `$ cat /etc/passwd | cat | cat`
  - `$ cat /etc/passwd | cat | cat > /tmp/p.txt`

File I/O versus Standard I/O

- File I/O (Unbuffered I/O)
  - Access via file descriptors
  - Talk to the kernel directly
- Standard I/O (Buffered I/O)
  - Access via wrapped file descriptors, i.e., the FILE data structure
  - Default wrappers for standard input, output, and errors
  - stdin, stdout, and stderr
  - The fileno function

Unbuffered I/O

- Definition
  - Each `read` or `write` invokes a system call in the kernel
  - Not buffered in user space programs and libraries
- Usually can be performed by using only the five functions
  - `open`, `read`, `write`, `lseek`, and `close`

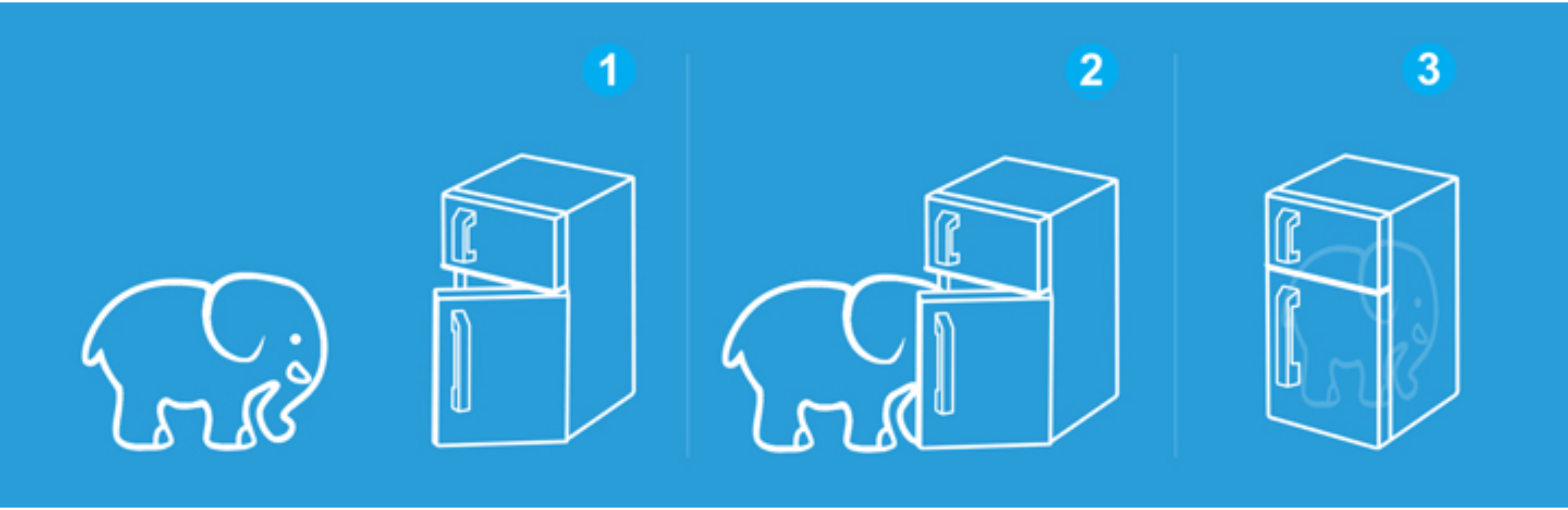
File Descriptors

- In the kernel, all opened files are referred to by file descriptors
- It is a non-negative integer
- A convention for shells and many applications
  - File descriptor 0, 1, and 2 refers to standard input, output, and error, respectively
  - `STDIN_FILENO` (0), `STDOUT_FILENO` (1), `STDERR_FILENO` (2)

    -- defined in the header file `unistd.h`
- The file descriptors can be used in a process is ranged from 0 to `OPEN_MAX`-1
  - Can be changed using the `setrlimit(2)` function
  - But it requires root permissions

File and Standard I/O

Open/create Files, and get a fd | Use fds to read/write/lseek files | Close files



File I/O Functions

The open(2) Function

- Open a file
- Synopsis
  - `int open(const char *pathname, int flags, mode_t mode);`
  - Returns: file descriptor opened for write-only if OK, -1 on error
- Mandatory flags
  - `O_RDONLY`
  - `O_WRONLY`
  - `O_RDWR`
- Common optional flags
  - `O_APPEND`
  - `O_CREAT`
  - `O_EXCL`
  - `O_TRUNC`
  - `O_SYNC`

The creat(2) Function

- Open a file for write only
- Synopsis
  - `int creat(const char *pathname, mode_t mode);`
  - Returns: file descriptor if OK, -1 on error
- It is equivalent to
  - `open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);`

The close(2) Function

- Close a opened file
- Synopsis
  - `int close(int filedes);`
  - Returns: 0 if OK, -1 on error
- When a process terminates, all of its open files are closed automatically

The lseek(2) Function

- Move the "file pointer" position
- Synopsis
  - `off_t lseek(int fd, off_t offset, int whence);`
  - Returns: new file offset if OK, -1 on error
  - Usually off_t is 32-bit long
  - Consider the `lseek64()` function using `off64_t`
- Choices of *whence*
  - `SEEK_SET, SEEK_CUR, SEEK_END`
- Can be used to determine if a file is seekable

Seeking Different File Types

```
#include "apue.h“   /* fig 3.1 */
int main(void) {
        if (lseek(STDIN_FILENO, 0, SEEK_CUR) == -1)
                printf("cannot seek\n");
        else
                printf("seek OK\n");
        exit(0);
}
```

```
$ ./seek < /etc/passwd
seek OK
$ cat < /etc/passwd | ./seek
cannot seek
$ ./seek < /var/spool/cron/FIFO
cannot seek
```

File Hole

- The `lseek(2)` and the `write(2)` function

```
#include "apue.h“  /* fig 3.2 */
#include <fcntl.h>
char   buf1[] = "abcdefghij“, 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, 16384, SEEK_SET) == -1)
        err_sys("lseek error");
    /* offset now = 16384 */
    if (write(fd, buf2, 10) != 10)
        err_sys("buf2 write error");
    /* offset now = 16394 */
    exit(0);
}
```

File Hole (Cont'd)

- The resulted file with a hole

```
$ ./fig3.2-hole
$ ls -l file.hole
-rw-r--r-- 1 chuang chuang 16394 2009-01-01 11:45 file.hole
$ hexdump -C file.hole
00000000  61 62 63 64 65 66 67 68  69 6a 00 00 00 00 00 00  |abcdefghij......|
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00004000  41 42 43 44 45 46 47 48  49 4a                    |ABCDEFGHIJ|
0000400a
```

- Compare with a no-hole file

```
$ ls -ls file.*
 8 -rw-r--r-- 1 chuang chuang 16394 2009-01-01 11:45 file.hole
20 -rw-r--r-- 1 chuang chuang 16394 2009-01-01 11:49 file.nohole
```

The read(2) Function

- Read from an opened file
- Synopsis
  - `ssize_t read(int fd, void *buf, size_t nbytes);`
  - Returns: number of bytes read, 0 if EOF, -1 on error
- File offset moves forward after read
- The number of read bytes <= the number of requested bytes, when would “<“ happen?
  - Regular file: A special case when *read* encounters EOF
  - Network: Depends on network buffering state
  - Pipe of FIFO: Read all available data
  - The read operation may be interrupt by signals

The write(2) Function

- Write data to an opened file
- Synopsis
  - `ssize_t write(int fd, const void *buf, size_t nbytes);`
  - Returns: number of bytes written if OK, -1 on error
- File offset moves forward after write

File I/O:
Other Issues

I/O Efficiency

- A simple "cat" program
  - How to choose the size of a buffer?

Hint: lsblk -o NAME,PHY-SeC

```
#include "apue.h"
#define   BUFFSIZE  16
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);
}
```

I/O Efficiency (Cont'd)

Hint: free && sync && echo 3 > /proc/sys/vm/drop_caches && free

- Command: `$ ./mycat < filename > /dev/null`

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

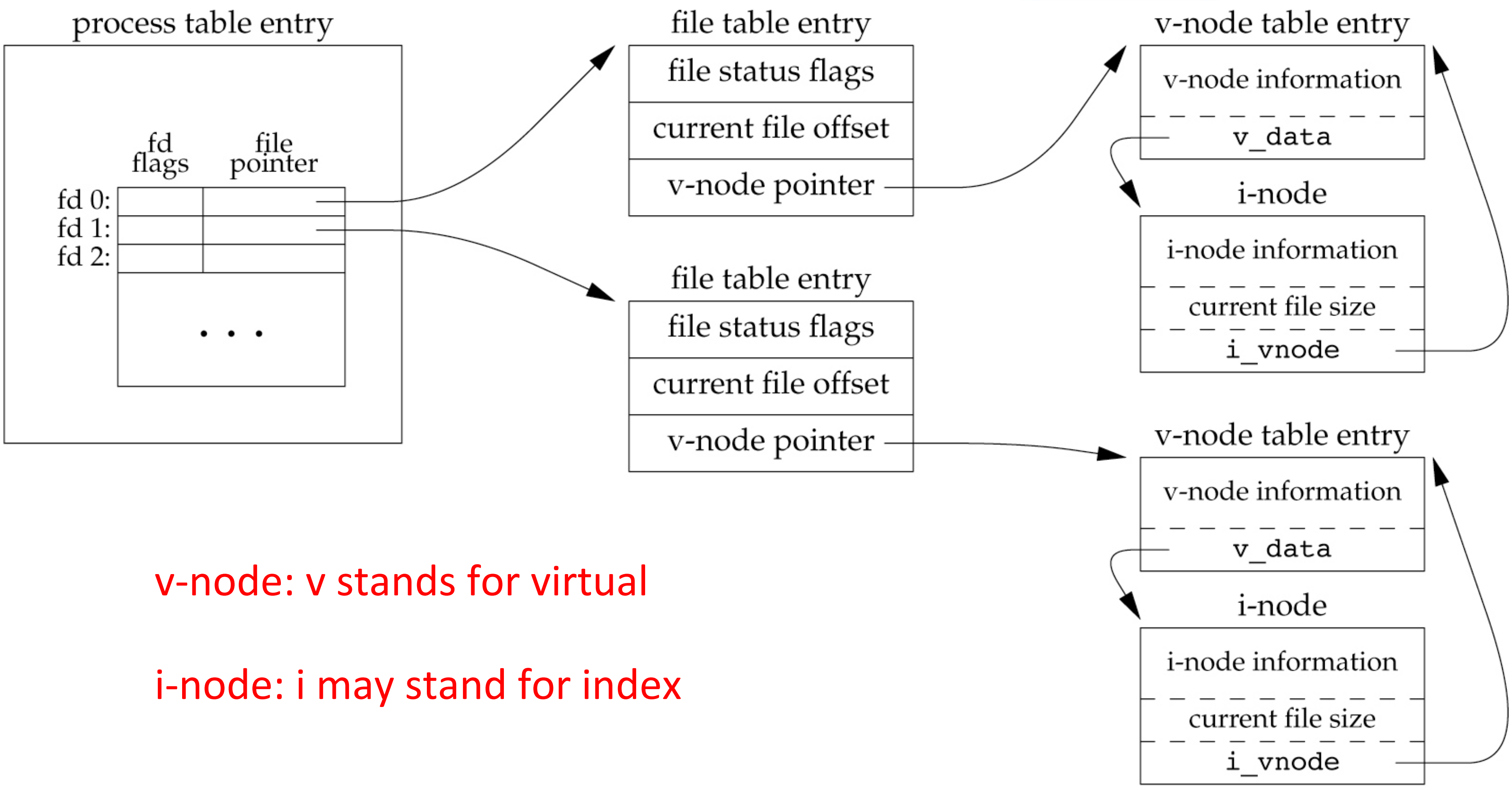
sync, fsync, And fdatasync Functions

- *Delayed write*
  - When data is copied to the kernel, it is queued for writing to disk at some later time
- Ask the kernel *starting* to write cached disk blocks
- For a specific file
  - `int fsync(int fd); /* filedata + metadata */`
  - `int fdatasync(int fd); /* filedata only */`
  - Return values for the above two functions: 0 if OK, -1 on error
- For all files ← but it returns immediately!!!
  - `void sync(void); /* filedata + metadata */`

File Sharing – An Overview

- An opened file can be shared among different processes
- The kernel maintains several different data structures for opened files
  - Each process has an entry in the *process table*
  - Each process table entry contains *a table of opened file descriptors*
  - A *file table* for all opened files
  - Each file table is associated with a *v-node structure*

File Sharing – Kernel Data Structures

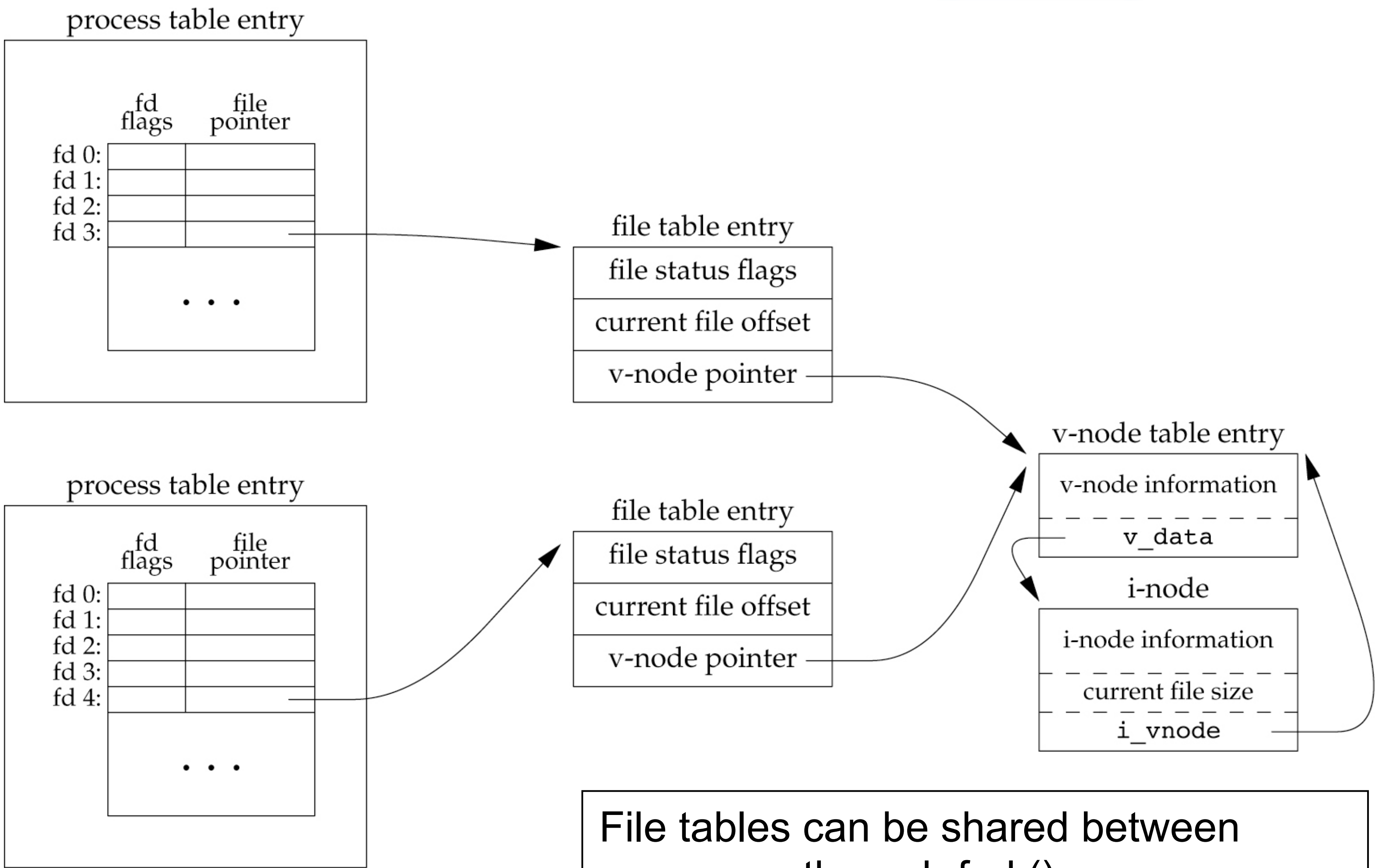


v-node: v stands for virtual

i-node: i may stand for index

Figure 3.7 Kernel data structures for open files

File Sharing – Open The Same File



File tables can be shared between processes through fork()

Figure 3.8 Two independent processes with the same file open

Atomic Operation

- Consider the following program

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

- What happens if two process do the same thing to the same file?
- Any operation that requires more than one function call is not atomic!

pread And pwrite

- Atomic seek and read/write
- Synopsis
  - `ssize_t pread(int fd, void *buf, size_t nbytes, off_t offset);`
  - Returns: number of bytes read, 0 if EOF, -1 on error
  - `ssize_t pwrite(int fd, const void *buf, size_t nbytes, off_t offset);`
  - Returns: number of bytes written if OK, -1 on error
- Seek first and then read or write
- There is no way to interrupt the two operations
- The current file offset is not updated

Atomic Creating of A Non-Existing File

- Create a file if it does not exist, legacy. Not atomic!

```
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");
    }
}
```

- Can be atomically done using open with `O_CREAT` and `O_EXCL`
  - `open(pathname, O_CREAT | O_EXCL, mode)`

dup/dup2 Functions

- dup: Duplicate a file descriptor
- dup2: Duplicate a file descriptor to a targeted descriptor

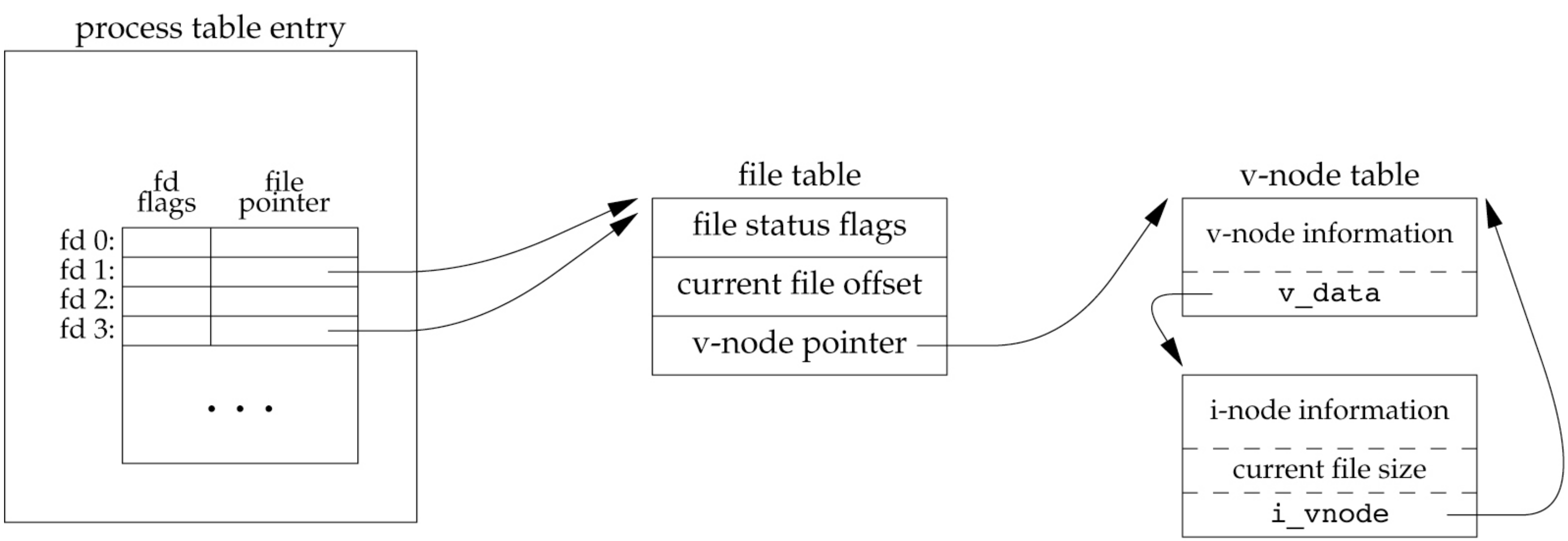


Figure 3.9 Kernel data structures after dup(1)

dup/dup2 Functions (Cont'd)

- Synopsis
  - `int dup(int fd);`
  - `int dup2(int fd, fd2);`
  - Returns: both return the new file descriptor if OK, -1 on error
- Equivalent operations
  - `dup(fd)`
    - `fcntl(fd, F_DUPFD, 0);`
  - `dup2(fd, fd2)`
    - ```
      close(fd2);
      fcntl(fd, F_DUPFD, fd2);
      ```
    - dup2 is an atomic operation
      ```

Why do We Need dup/dup2 ???

- I think there are two reasons
  - stdin/stdout/stderr redirections
  - copy the fds for the child processes
- Example:

```
printf("stdout, ");
  ....
  fd = dup(fileno(stdout));
  freopen("stdout.out", "w", stdout);
  printf("stdout in file\n");
  ...
  dup2(fd, fileno(stdout));
  printf("stdout again\n");
  ....
```

The fcntl Function

- Change the properties of an opened file
- Synopsis
  - `int fcntl(int fd, int cmd, … /* int arg */);`
  - Returns: depends on cmd if OK, -1 on error
- Common commands
  - `F_DUPFD` – duplicate the file descriptor
  - `F_GETFD/F_SETFD` – get or set the file descriptor flag
    - supports only FD_CLOEXEC (close-on-exec) ← so that the child processes won't get the FDs…
  - `F_GETFL/F_SETFL` – get or set the file status flags
    - `O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_NONBLOCK, O_SYNC, …`

What's the difference between file descriptor flags and file status flags?

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

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

    switch (val & O_ACCMODE) {
    case O_RDONLY:
        printf("read only");
        break;

    case O_WRONLY:
        printf("write only");
        break;

    case O_RDWR:
        printf("read write"
        break;

    default:
        err_dump("unknown a
    }

    if (val & O_APPEND)
        printf(", append");
    if (val & O_NONBLOCK)
        printf(", nonblocki
    if (val & O_SYNC)
        printf(", synchrono

#if !defined(_POSIX_C_SOURC
    if (val & O_FSYNC)
        printf(", synchrono
#endif

    putchar('\n');
    exit(0);
}
```

The messy bitmaps we discussed earlier!

Try these:

- ./fileflags 0 < /dev/tty
- ./fileflags 1 > /tmp/out.txt
- cat !$
- ./fileflags 1 >> /tmp/out.txt
- cat !$
- ./fileflags 5 5<>/tmp/tmp.txt

Bit Manipulations

- Three popular operands
  - >> is the arithmetic (or signed) right shift operator.
  - >>> is the logical (or unsigned) right shift operator.
  - << is the left shift operator, and meets the needs of both logical and arithmetic shifts.
- Example:
  - # define CIA_CTRL_PCI_EN (1 << 0)
  - # define CIA_CTRL_PCI_LOCK_EN (1 << 1)
  - # define CIA_CTRL_PCI_LOOP_EN (1 << 2)

The fcntl Function – Change Status Flags

- `val |= flags;`
- `val &= ~flags;`

Why do I show you this code snippet?

```
#include "apue.h"
#include <fcntl.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");
}
```

The ioctl(2) Function

- The ultimate function to control all I/O operations
- Synopsis
  - `int ioctl(int fd, int request, …);`
  - Returns: -1 on error, something else if OK
- *There is no standard for the `ioctl` function*
  - Each device driver can define its own set of `ioctl` commands, a common method to handle user-kernel interactions
- The *request* is a device dependent request code
- The third argument is usually an untyped pointer to memory

Sample of ioctl Function

```
static int
e100_ioctl(struct net_device *dev, struct ifreq *ifr, int cmd)
{
   struct mii_ioctl_data *data = if_mii(ifr);
   struct net_local *np = netdev_priv(dev);
   int rc = 0;
    int old_autoneg;

   spin_lock(&np->lock); /* Preempt protection */
   switch (cmd) {
      /* The ioctls below should be considered obsolete but are */
      /* still present for compatibility with old scripts/apps  */
      case SET_ETH_SPEED_10:                  /* 10 Mbps */
         e100_set_speed(dev, 10);
         break;
      case SET_ETH_SPEED_100:                 /* 100 Mbps */
         e100_set_speed(dev, 100);
         break;
```

/dev/fd

- Modern systems provide a `/dev/fd` directory
- It is a virtual file system
- Each process has its own view of `/dev/fd`
- Assume that descriptor *n* has been opened, open the file `/dev/fd/n` is equivalent to duplicate descriptor n
  - `fd = open("/dev/fd/0", mode);`
  - `fd = dup(0);`
- The main usage of the `/dev/fd` files is from the shell
- Allow programs to handle standard input and standard output as regular files

Compare:

- cat /etc/passwd | cat –
- cat /etc/passwd | cat /dev/fd/0

STANDARD I/O FUNCTIONS

Standard (Buffered) I/O

- Standard I/O handles … ← different from file I/O
  - Buffer allocation
  - Perform I/O in optimal-sized chunks
- Standard I/O is easier to use
- The FILE structure
  - Treat all opened files as a stream
  - Associate the stream with an underlying file descriptor ← what we learned last week
  - Maintain buffer states

Buffering

- Fully buffered
  - Files residing on disk are normally fully buffered by the standard I/O library
  - The buffer used is usually obtained by one of the standard I/O functions calling *malloc* the first time I/O is performed on a stream ← before calling the malloc, the pointer == NULL
- Line buffered
  - the standard I/O library performs I/O when a newline character is encountered on input or output
  - Caveats
    - Buffer size is limited – I/O may be performed before a newline
    - Before read, all line-buffered output streams are flushed
- Unbuffered

Default Buffering Modes

- ISO C: the standard
  - Standard input and standard output are fully buffered, if and only if they *do not refer to an interactive device*
  - Standard error is never fully buffered ← Why?
- Most implementations follow the below convention:
  - Standard error is always unbuffered
  - All other streams are line buffered if they refer to a terminal device; otherwise, they are fully buffered

Functions for Setting Buffer

- Synopsis
  - *void setbuf(FILE *fp, char *buf);*
    - *buf* must be the size of *BUFSIZ*
  - *int setvbuf(FILE *fp, char *buf, int mode, size_t size);*
  - Returns: 0 if OK, nonzero on error
  - Need to be done before the first file access
- Buffering is disabled if *buf* is NULL
- Buffering mode
  - _IOFBF: fully buffered
  - _IOLBF: line buffered
  - _IONBF: unbuffered

Open Files

- Open files
  - *FILE* fopen(char *pathname, char *mode);*
  - *FILE *freopen(char *pathname, char *mode, FILE *fp);*
  - *FILE *fdopen(int fd, char *mode);*
  - Returns: file pointer if OK, NULL on error
- modes
  - ***r*** or ***rb***: open for reading
  - ***w*** or ***wb***: truncate to 0 length or create for writing
  - ***a*** or ***ab***: append, open for writing at EOF or create for writing
  - ***r+***, ***r+b***, or ***rb+***: open for reading and writing
  - ***w+***, ***w+b***, or ***wb+***: equivalent to w or wb plus reading
  - ***a+***, ***a+b***, or ***ab+***: equivalent to a or ab plus reading
  - Note: UNIX does not require ***t*** (text) mode for text files

Read and Write a String – By Character

- Read
  - *int getc(FILE *fp);*   *int fgetc(FILE *fp);*
  - *int getchar(void);*
  - Returns: next character if OK, *EOF* on EOF or error
- How to tell EOF or error?
  - *int ferror(FILE *fp); int feof(FILE *fp);*
  - Returns: nonzero (true) if a condition is true, or zero (false) otherwise
- Write
  - *int putc(int c, FILE *fp); int fputc(int c, FILE *fp);*
  - *int putchar(int c);*
  - Returns: c if OK, *EOF* on error

- getc (…) and putc(...) can be implemented as macros

Macro versus Function Calls

- Arguments of getc(…) should not have any side effects, cuz it may be evaluated multiple times
- We can pass fgetc(...) as a function pointer to other functions
- Calling fgetc(...) probably takes longer

Reading Char-by-Char: Example

```
#include "apue.h"

int
main(void)
{
    int     c;

    while ((c = getc(stdin)) != EOF)
        if (putc(c, stdout) == EOF)
            err_sys("output error");

    if (ferror(stdin))
        err_sys("input error");

    exit(0);
}
```

What does this function do?

Figure 5.4 Copy standard input to standard output using `getc` and `putc`

Read and Write a String – By Line

- Read
  - *char *fgets(char *buf, int n, FILE *fp);*
  - *char *gets(char *buf);*
  - Returns: buf if OK, NULL on end of file or error

Using gets(…) is a bad idea, why?

- Write
  - *int fputs(char *str, FILE *fp);*
  - *int puts(char *str);*
  - Returns: non-negative value if OK, EOF on error

Reading Line-by-Line: Example

```
#include "apue.h"

int
main(void)
{
    char    buf[MAXLINE];

    while (fgets(buf, MAXLINE, stdin) != NULL)
        if (fputs(buf, stdout) == EOF)
            err_sys("output error");

    if (ferror(stdin))
        err_sys("input error");

    exit(0);
}
```

Figure 5.5 Copy standard input to standard output using `fgets` and `fputs`

Standard I/O Efficiency

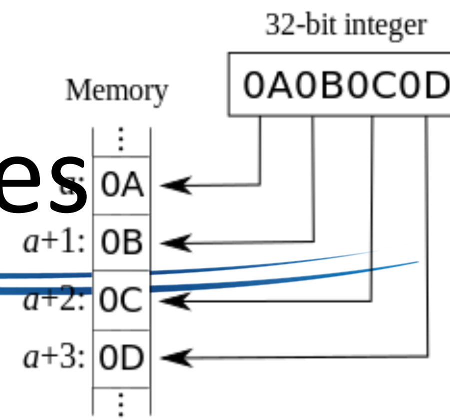
- Performance of reading a 98.5MB file from stdin (roughly 3 million lines) and writing to stdout (/dev/null)

Function	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	Bytes of program text
best time from Figure 3.6	0.05	0.29	3.18	
`fgets, fputs`	2.27	0.30	3.49	143
`getc, putc`	8.45	0.29	10.33	114
`fgetc, fputc`	8.16	0.40	10.18	114
single byte time from Figure 3.6	134.61	249.94	394.95	

Figure 5.6 Timing results using standard I/O routines

Binary I/O

- Synopsis
  - *size_t fread(void *ptr, size_t size, size_t nobj, FILE *fp);*
  - *size_t fwrite(void *ptr, size_t size, size_t nobj, FILE *fp);*
  - Returns: number of objects read or written
- Read/write multiple objects in each invocation



Binary I/O – Examples

- Example #1

```
float data[10];
if (fwrite(&data[2], sizeof(float), 4, fp) != 4)
      err_sys("fwrite error");
```

- Example #2

```
struct {   short      count;
      longtotal;
      charname[NAMESIZE];
} item;
if (fwrite(&item, sizeof(item), 1, fp) != 1)
      err_sys("fwrite error");
```

- Notes
  - The offset of a member within a structure can differ between compilers and systems ← aligned or not for speed versus space
  - The binary formats used to store multibyte integers and floating-point values differ among machine architectures ← big/little endian

Positioning a Stream

- Similar to *seek* …
  - *int fseek(FILE *fp, long offset, int whence);*
  - *long ftell(FILE *fp);*
  - *void rewind(FILE *fp);*
- Similar functions with offset *off_t* ← why?
  - *off_t ftello(…)* and *int fseeko (….)*
- ISO C standard: *fgetpos(…)* and *fsetpos(…),* better for porting to non-UNIX systems

Temporary Files

- Create a temporary file
- Synopsis
  - *char *tmpnam(char *ptr);*
  - Returns: pointer to unique pathname
  - *FILE *tmpfile(void);*
  - Returns: file pointer if OK, NULL on error
- It is not recommend to use *tmpnam*
  - It uses a static buffer to store generated filename ← something may happen between calling tmpnam and open file…
  - As the generated name are /tmp/fileXXXXXX, it might be guessed
  - Solutions
    - Use *tmpfile* or *mkstemp* instead
    - Open the temporary file using open(2) with the O_EXCL flag

Wha le?

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

void make_temp(char *template);

int
main()
{
    char    good_template[] = "/tmp/dirXXXXXX"; /* right way */
    char    *bad_template = "/tmp/dirXXXXXX";   /* wrong way*/

    printf("trying to create first temp file...\n");
    make_temp(good_template);
    printf("trying to create second temp file...\n");
    make_temp(bad_template);
    exit(0);
}

void
make_temp(char *template)
{
    int         fd;
    struct stat sbuf;

    if ((fd = mkstemp(template)) < 0)
        err_sys("can’t create temp file");
    printf("temp name = %s\n", template);
    close(fd);
    if (stat(template, &sbuf) < 0) {
        if (errno == ENOENT)
            printf("file doesn’t exist\n");
        else
            err_sys("stat failed");
    } else {
        printf("file exists\n");
        unlink(template);
    }
}
```

Figure 5.13 Demonstrate `mkstemp` function

Reading Assignments

- Chapter 3: File I/O
- Chapter 5: Standard I/O Library

QUESTION?

Assignment #2 (5%)

- Write your own dup2 function that behaves the same way as the dup2 function described in Section 3.12, without calling the fcntl function. Be sure to handle errors correctly.
- Hint: If fcntl cannot be invoked, you will have to use dup. Then you have no control over which file descriptor will be used by the dup function call. Try to design a workaround of this.
- Submission:
  - (1%) Submit you pseudocode in plan-text, with the file name: hw02_[YourStudentID].txt
  - (3%) Submit your code with the file name: hw02_[YourStudentID].c . You get 3 points once your code can handle normal test cases prepared by TA.
  - (1%) You get one more point if you handle all the errors correctly.
- Due date: Oct 4th