Unix programming interface for file I/O operations and pipes

M1 MOSIG – Operating System Design

Renaud Lachaize

Acknowledgments

- Many ideas and slides in these lectures were inspired by or even borrowed from the work of others:
  - Arnaud Legrand, Noël De Palma, Sacha Krakowiak
  - David Mazières (Stanford)
    - (many slides/figures directly adapted from those of the CS140 class)
  - Remzi and Andrea Arpaci-Dusseau (U. Wisconsin)
  - Randall Bryant, David O'Hallaron, Gregory Kesden, Markus Püschel (Carnegie Mellon University)
    - Textbook: Computer Systems: A Programmer's Perspective (2nd Edition) a.k.a. "CSAPP"
    - CS 15-213/18-243 classes (**many slides/figures directly adapted from these classes**)
  - Textbooks (Silberschatz et al., Tanenbaum)

Outline

- Introduction
- Basic Unix I/O interface
  - Main primitives
  - Kernel management of open files
- Unix standard I/O interface
- Inter-process communication via pipes and FIFOs
- Dealing with short counts – an example : the RIO library
- Wrap-up on Unix I/O interfaces

Unix files

- A Unix *file* is a sequence of *m* bytes:
  - $B_0, B_1, \ldots., B_k, \ldots., B_{m-1}$

- All I/O devices are represented as files:
  - **`/dev/sda2`** (**`/usr`** disk partition)
  - **`/dev/tty2`** (terminal)

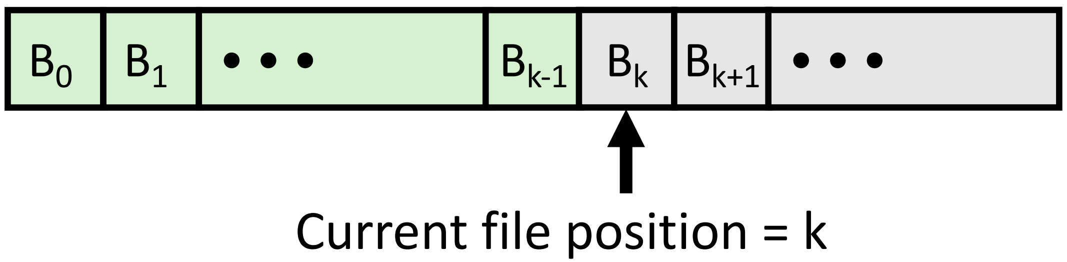
- Even the kernel sometimes represented as a file:
  - **`/dev/kmem`** (kernel memory image)
  - **`/proc`** (kernel data structures)

Unix file types

- Regular file
  - File containing user/app data (binary, text, whatever)
  - OS does not know anything about the format
    - Other than “sequence of bytes”, akin to main memory
- Directory file
  - A file that contains the names and locations of other files
- Character special and block special files
  - Terminals (character special) and disks (block special)
- FIFO (named pipe)
  - A file type used for inter-process communication (details later)
- Socket
  - A file type used for network communication between processes

Unix I/O

- Key Features
  - Elegant mapping of files to devices allows kernel to export simple interface called Unix I/O
  - Important idea: All input and output is handled in a consistent and uniform way
- Basic Unix I/O operations (system calls):
  - Opening and closing files
    - **`open()`** and **`close()`**
  - Reading and writing a file
    - **`read()`** and **`write()`**
  - Changing the ***current file position*** (seek)
    - indicates next offset into file to read or write
    - **`lseek()`**



Opening files

- Opening a file informs the kernel that you are getting ready to access that file

```
int fd;   /* file descriptor */

if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {
   perror("open");
   exit(1);
}
```

- Returns a small identifying integer *file descriptor*
  - **`fd == -1`** indicates that an error occurred
- Each process created by a Unix shell begins life with three open files associated with a terminal:
  - 0: standard input
  - 1: standard output
  - 2: standard error

Closing files

- Closing a file informs the kernel that you are finished accessing that file

```
int fd;     /* file descriptor */
int retval; /* return value */

if ((retval = close(fd)) < 0) {
   perror("close");
   exit(1);
}
```

- Closing an already closed file is a recipe for disaster in threaded programs (more details on this later)

- Moral: Always check return codes, even for seemingly benign functions such as **`close()`**

Reading files

- Reading a file copies bytes from the current file position to memory, and then updates file position

```
char buf[512];
int fd;        /* file descriptor */
int nbytes;    /* number of bytes read */

/* Open file fd ...  */
/* Then read up to 512 bytes from file fd */
if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {
   perror("read");
   exit(1);
}
```

- Returns number of bytes read from file **`fd`** into **`buf`**
  - Return type **`ssize_t`** is signed integer (unlike **`size_t`**)
  - **`nbytes < 0`** indicates that an error occurred
  - ***Short counts*** (**`nbytes < sizeof(buf)`**) are possible and are not errors!

Writing files

- Writing a file copies bytes from memory to the current file position, and then updates current file position

```
char buf[512];
int fd;         /* file descriptor */
int nbytes;     /* number of bytes read */

/* Open the file fd ... */
/* Then write up to 512 bytes from buf to file fd */
if ((nbytes = write(fd, buf, sizeof(buf)) < 0) {
   perror("write");
   exit(1);
}
```

- Returns number of bytes written from **`buf`** to file **`fd`**
  - **`nbytes < 0`** indicates that an error occurred
  - As with reads, short counts are possible and are not errors!

Simple Unix I/O example

- Copying standard input to standard output, one byte at a time

```
int main(void)
{
    char c;
    int len;

    while ((len = read(0 /*stdin*/, &c, 1)) == 1) {
        if (write(1 /*stdout*/, &c, 1) != 1) {
            exit(20);
        }
    }
    if (len < 0) {
        printf (“read from stdin failed”);
        exit (10);
    }
    exit(0);
}
```

File metadata

- *Metadata* is data about data, in this case file data
- Per-file metadata maintained by kernel
  - accessed by users with the **`stat`** and **`fstat`** functions

```
/* Metadata returned by the stat and fstat functions */
struct stat {
    dev_t          st_dev;      /* device */
    ino_t          st_ino;      /* inode */
    mode_t         st_mode;     /* protection and file type */
    nlink_t        st_nlink;    /* number of hard links */
    uid_t          st_uid;      /* user ID of owner */
    gid_t          st_gid;      /* group ID of owner */
    dev_t          st_rdev;     /* device type (if inode device) */
    off_t          st_size;     /* total size, in bytes */
    unsigned long  st_blksize;  /* blocksize for filesystem I/O */
    unsigned long  st_blocks;   /* number of blocks allocated */
    time_t         st_atime;    /* time of last access */
    time_t         st_mtime;    /* time of last modification */
    time_t         st_ctime;    /* time of last change */
};
```

Example of accessing file metadata

```
/* statcheck.c - Querying and manipulating a file's meta data */
#include "csapp.h"

int main (int argc, char **argv)
{
    struct stat stat;
    char *type, *readok;

    Stat(argv[1], &stat);
    if (S_ISREG(stat.st_mode))
        type = "regular";
    else if (S_ISDIR(stat.st_mode))
        type = "directory";
    else
        type = "other";
    if ((stat.st_mode & S_IRUSR)) /* OK to read?*/
        readok = "yes";
    else
        readok = "no";

    printf("type: %s, read: %s\n", type, readok);
    exit(0);
}
```

```
unix> ./statcheck statcheck.c
type: regular, read: yes
unix> chmod 000 statcheck.c
unix> ./statcheck statcheck.c
type: regular, read: no
unix> ./statcheck ..
type: directory, read: yes
unix> ./statcheck /dev/kmem
type: other, read: yes
```

Repeated slide: opening files

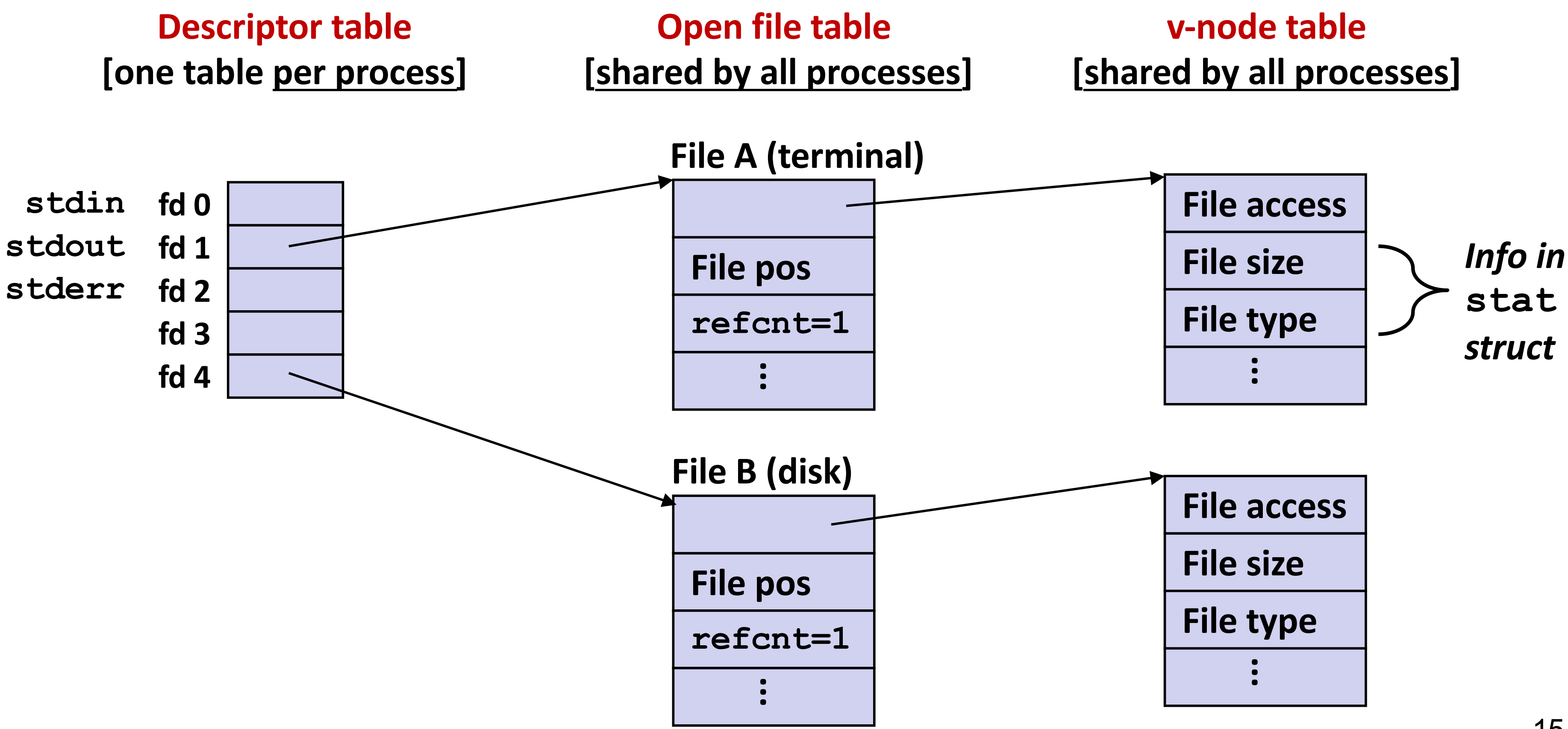
- Opening a file informs the kernel that you are getting ready to access that file

```
int fd;   /* file descriptor */

if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {
   perror("open");
   exit(1);
}
```

- Returns a small identifying integer *file descriptor*
  - **`fd == -1`** indicates that an error occurred
- Each process created by a Unix shell begins life with three open files associated with a terminal:
  - 0: standard input
  - 1: standard output
  - 2: standard error

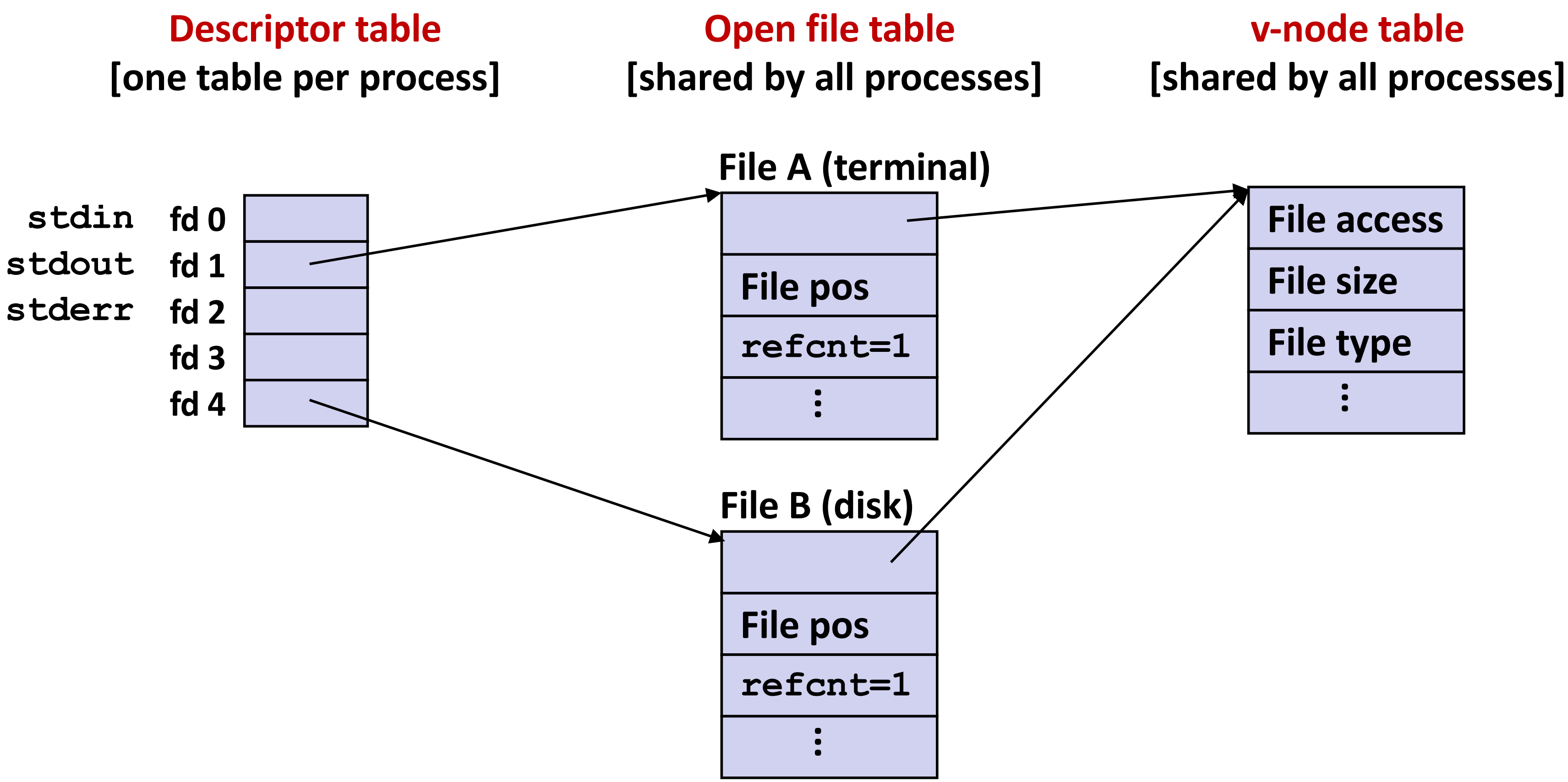
How a Unix kernel represents open files

- Two descriptors referencing two distinct open disk files.
- Descriptor 1 (stdout) points to terminal, and descriptor 4 points to open disk file



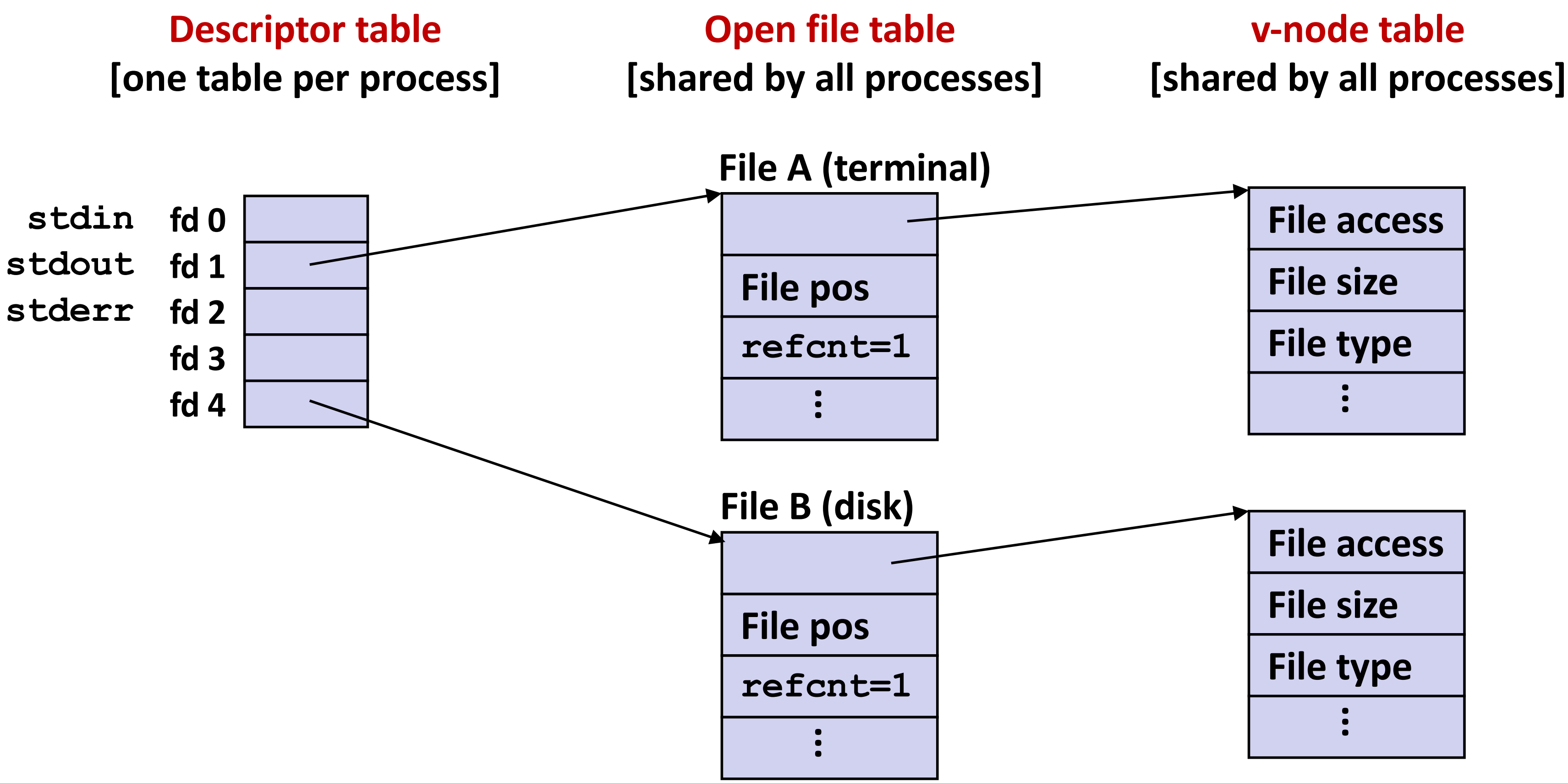
File sharing

- Two distinct descriptors sharing the same disk file through two distinct open file table entries
  - E.g., Calling **`open`** twice with the same **`filename`** argument



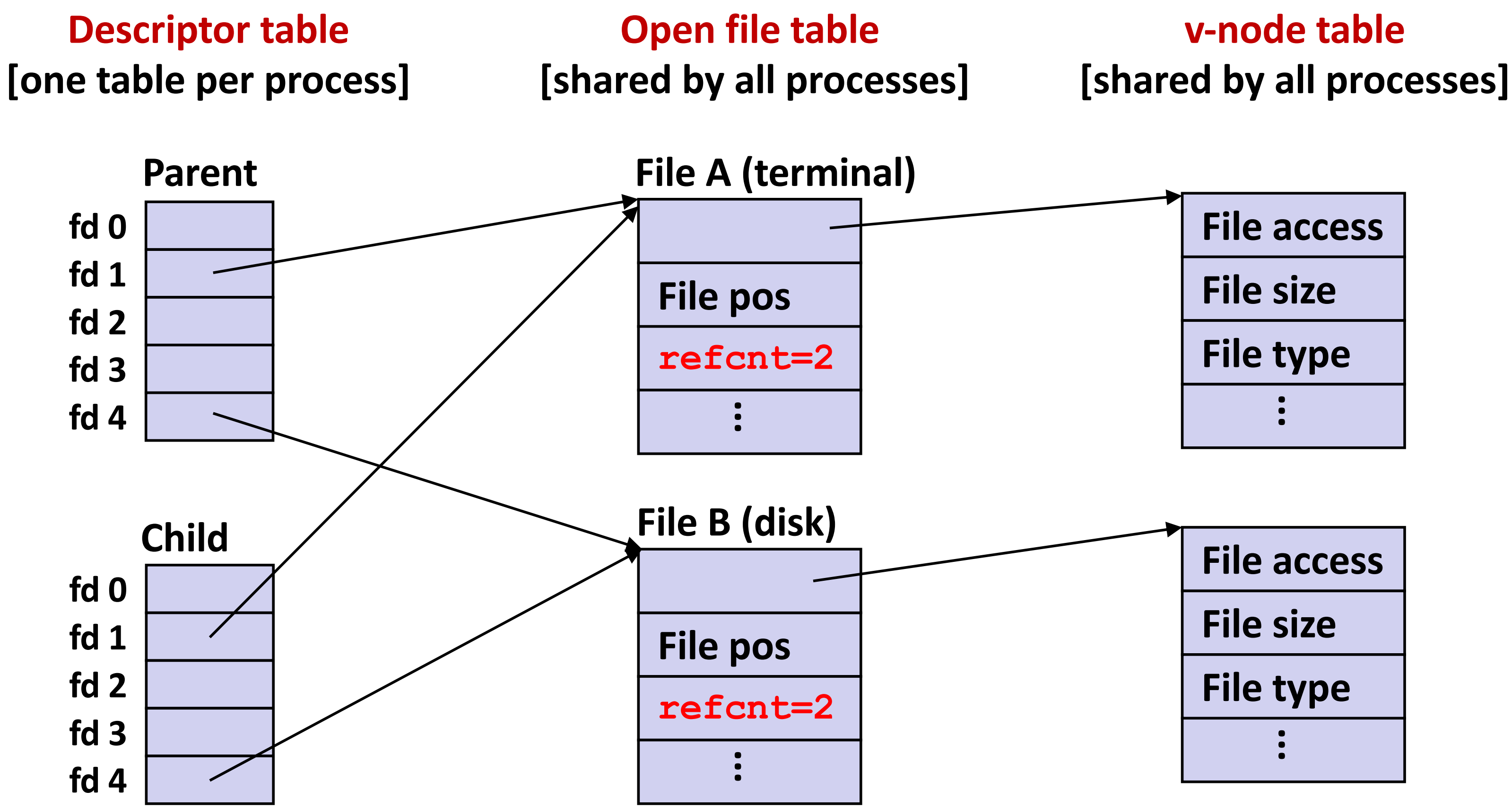
How processes share files
What happens upon fork

- A child process inherits its parent's open files
  - Note: situation unchanged by **`exec`** functions
- *Before* **`fork`** call:



How processes share files
What happens upon fork

- A child process inherits its parent's open files
- *After* **`fork`**:
  - Child's table same as parents, and +1 to each **`refcnt`** (reference counter)



I/O redirection

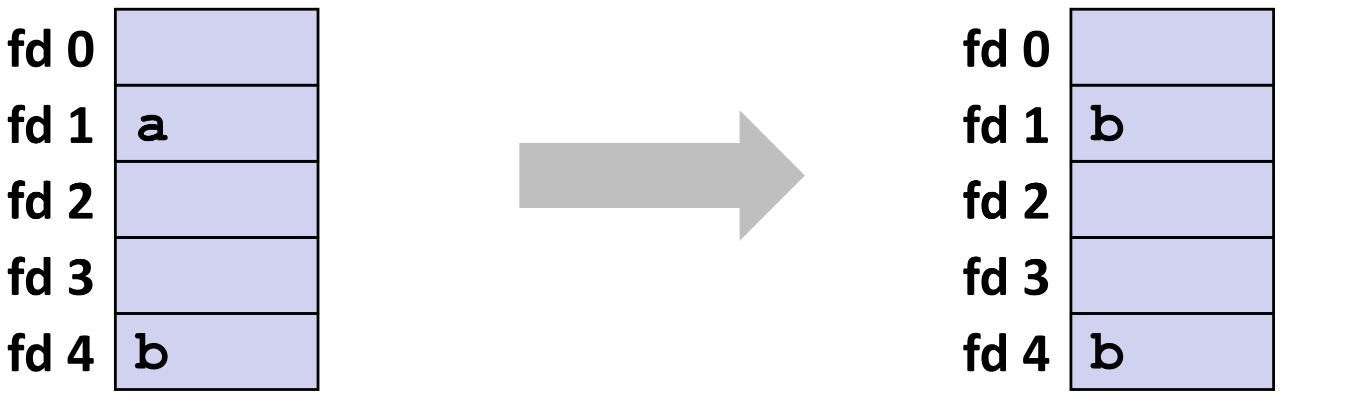
- Question: How does a shell implement I/O redirection?

  ```
  ls > foo.txt
  ```

- Answer: By calling the **`dup2(oldfd, newfd)`** function
  - Copies (per-process) descriptor table entry **`oldfd`** to entry **`newfd`**

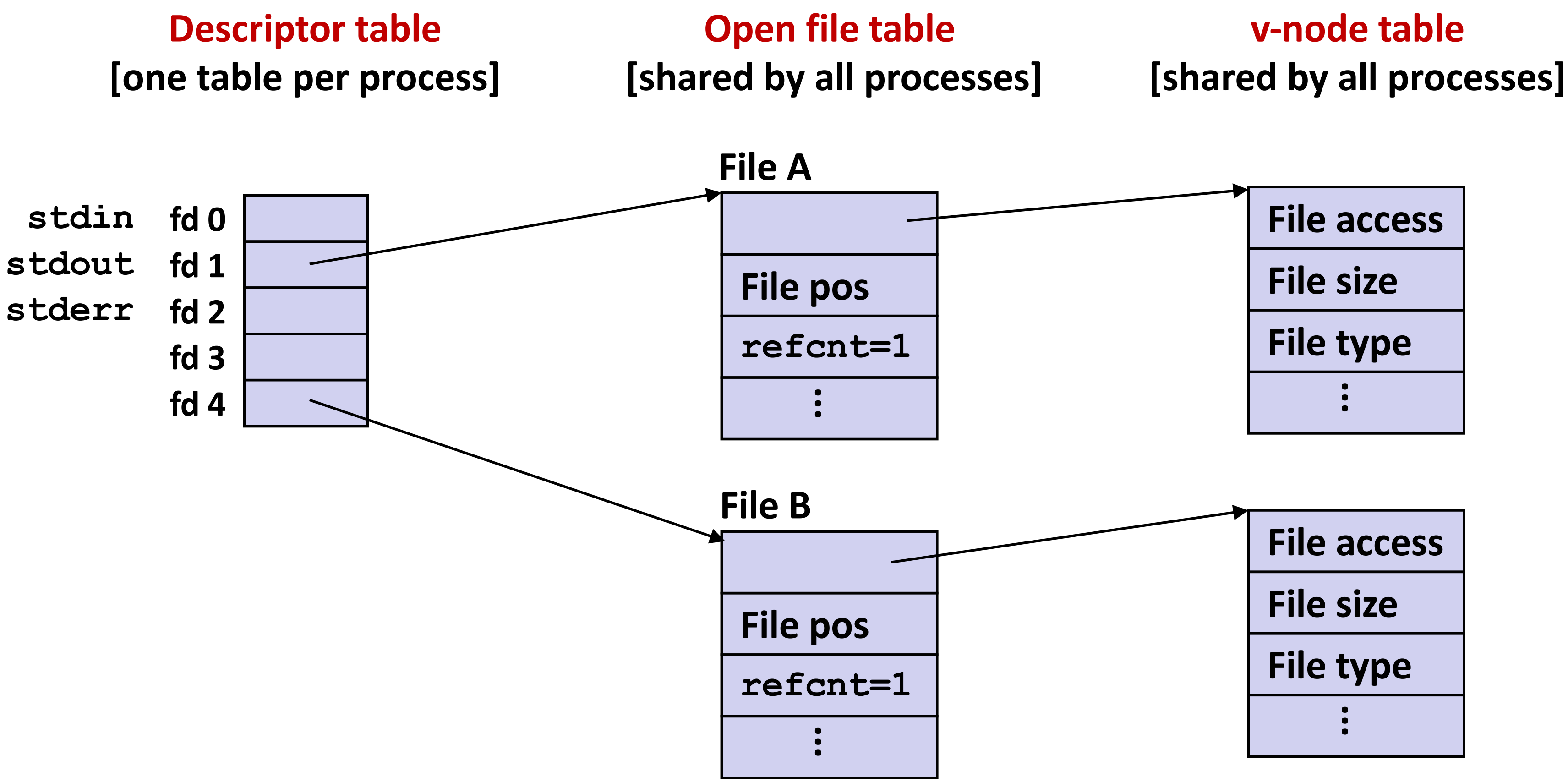
Descriptor table
before **`dup2(4,1)`**

Descriptor table
after **`dup2(4,1)`**



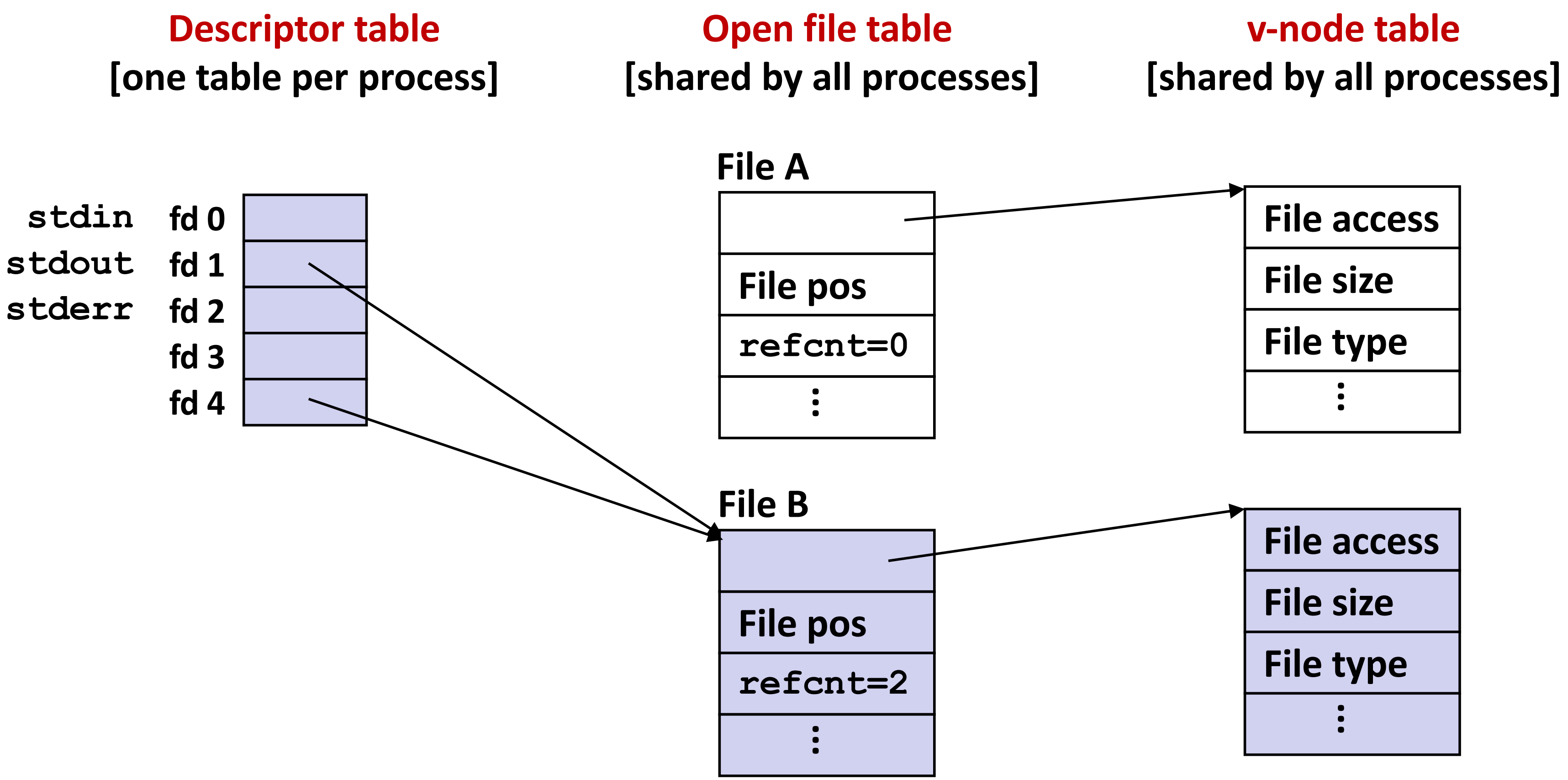
I/O redirection example

- Step #1: open file to which stdout should be redirected
  - Happens in child executing shell code, before calling **`exec`**



I/O redirection example (continued)

- Step #2: call **`dup2(4,1)`**
  - causes fd=1 (stdout) to refer to disk file pointed at by fd=4
  - (then fd=4 can be closed)



Outline

- Introduction
- Basic Unix I/O interface
  - Main primitives
  - Kernel management of open files
- Unix standard I/O interface
- Inter-process communication via pipes and FIFOs
- Dealing with short counts – an example : the RIO library
- Wrap-up on Unix I/O interfaces

Standard I/O functions

- The C standard library (**`libc`**) contains a collection of higher-level *standard I/O* functions

- Examples:
  - Opening and closing files (**`fopen`** and **`fclose`**)
  - Reading and writing bytes (**`fread`** and **`fwrite`**)
  - Reading and writing text lines (**`fgets`** and **`fputs`**)
  - Formatted reading and writing (**`fscanf`** and **`fprintf`**)

Standard I/O streams

- Standard I/O models open files as *streams*
  - Abstraction for a file descriptor and a buffer in user memory.
- C programs begin life with three open streams (defined in **stdio.h**)
  - **stdin** (standard input)
  - **stdout** (standard output)
  - **stderr** (standard error)

```
#include <stdio.h>
extern FILE *stdin;  /* standard input  (descriptor 0) */
extern FILE *stdout; /* standard output (descriptor 1) */
extern FILE *stderr; /* standard error  (descriptor 2) */

int main() {
    fprintf(stdout, "Hello, world\n");
}
```

Standard I/O streams (continued)

- Bridging streams and file descriptors
  - `FILE* fdopen(int fd, const char *mode);`

    Creates a stream from an existing file descriptor

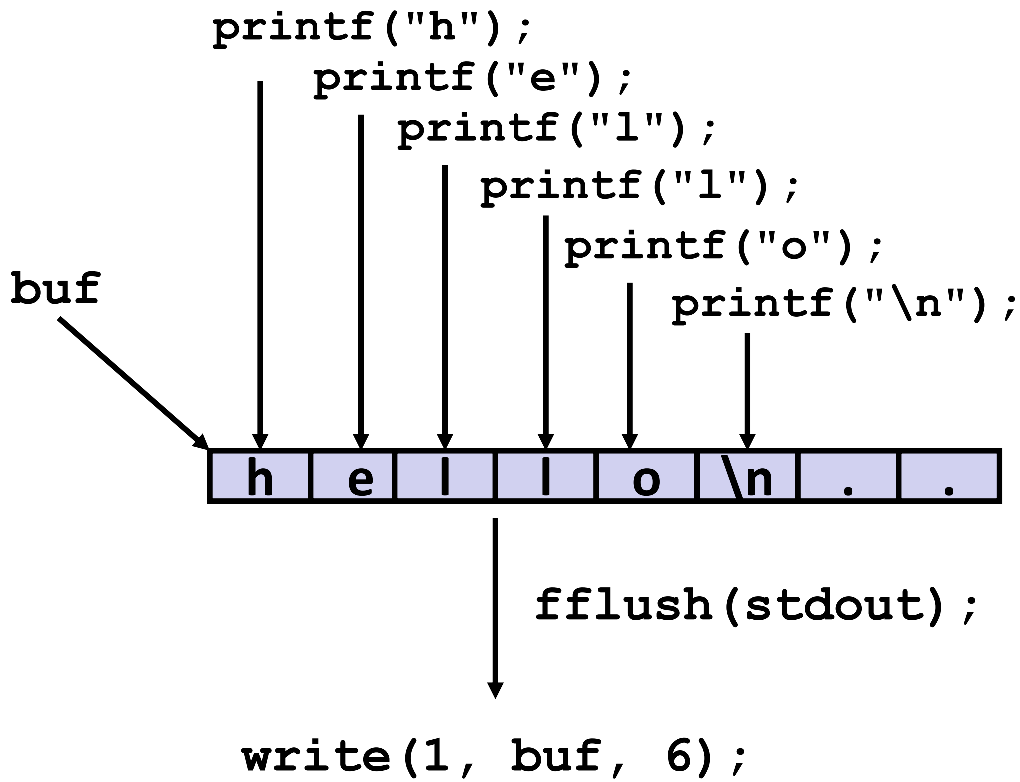
  - `int fileno(FILE *stream);`

    Returns the underlying file descriptor number of a given stream

  - Standard streams
    - Stream **`stdin`** associated with descriptor **`STDIN_FILENO`** (0)
    - Stream **`stdout`** associated with descriptor **`STDOUT_FILENO`** (1)
    - Stream **`stderr`** associated with descriptor **`STDERR_FILENO`** (2)

Buffering in standard I/O

- Standard I/O functions use buffered I/O



- Buffer flushed to output fd on "**`\n`**" or **`fflush`** call

Standard I/O buffering in action

- You can see this buffering in action for yourself, using the Unix **`strace`** program:

```
#include <stdio.h>

int main()
{
    printf("h");
    printf("e");
    printf("l");
    printf("l");
    printf("o");
    printf("\n");
    fflush(stdout);
    exit(0);
}
```

```
linux> strace ./hello
execve("./hello", ["hello"], [/* ... */]).
...
write(1, "hello\n", 6...)                  = 6
...
_exit(0)                                   = ?
```

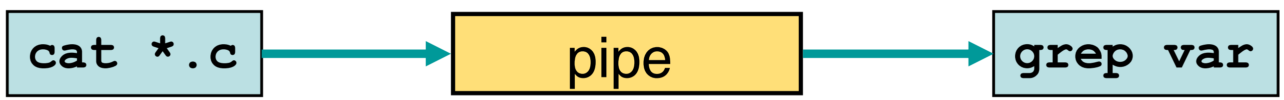
- Note: the general principle of I/O buffering is further explained in another part of the lecture (see the section about the RIO library)

Outline

- Introduction
- Basic Unix I/O interface
  - Main primitives
  - Kernel management of open files
- Unix standard I/O interface
- Inter-process communication via pipes and FIFOs
- Dealing with short counts – an example : the RIO library
- Wrap-up on Unix I/O interfaces

Unix pipes

- Pipes are a mechanism for inter-process communication (IPC)
- A pipe is essentially a (unidirectional) buffer that can be used for data exchange between a producer process and a consumer process
- Available at two levels: command line interface and programmatic interface
- Command line interface (shell)
  - Example : **`cat *.c | grep var`**
    - Creates two processes: P1 running **`cat *.c`** and P2 running **`grep var`**
    - Connects (redirects) P1’s standard output to the pipe’s input and the pipe’s output to P2’s standard input



Unix pipes
Programmatic interface

- User programs (not just shells) can create and interact with pipes through system calls
- A pipe is seen as a special kind of file
- The only way to share a pipe between processes is through inheritance of open files
- Typical usages:
  - Parent creates pipe then creates child then communicates with child through pipe (see following example)
  - Parent creates pipe, then create child1 and child2, then child1 and child2 communicate through pipe

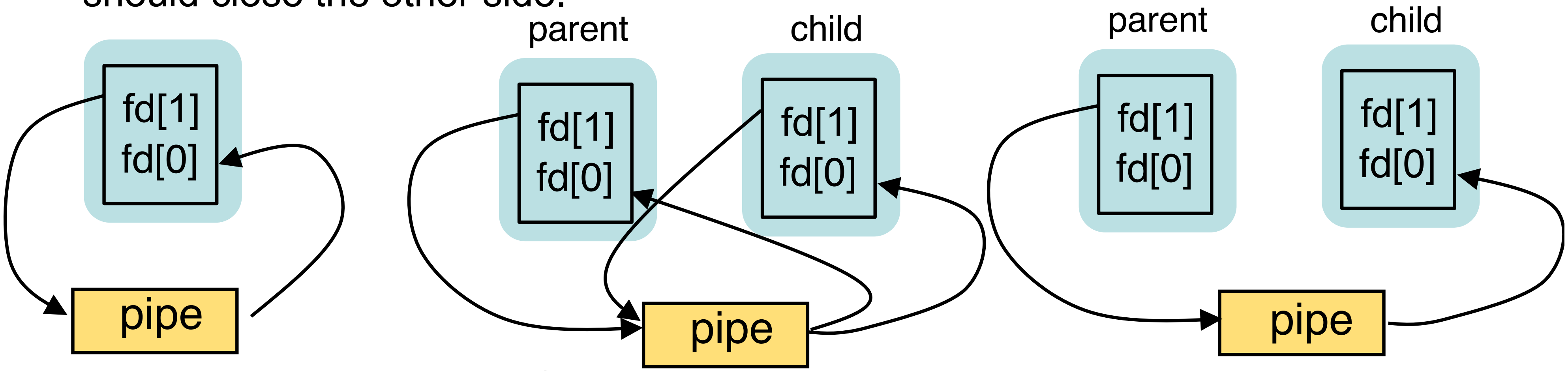
Unix pipes
Programmatic interface (continued)

Pipe creation: **`int pipe(int filedes[2])`**

```
int fd[2]; pipe(fd);
```

If the call succeeds, a pipe is created and the **`fd`** array is updated with the file descriptors of the **pipe's output** (in **`fd[0]`**) and the **pipe's input** (in **`fd[1]`**)
If the call fails, -1 is returned.

The pipe can then be transmitted through inheritance and used for communication. Each process will typically use only one side of the pipe and should close the other side.



after **`pipe(fd)`**

after **`fork()`** (copied descriptors)

after closing unused descriptors

Unix pipe example

```
#include ...
#define BUFSIZE 10
int main(void) {
  char bufin[BUFSIZE] = "empty";
  char bufout[BUFSIZE] = "hello";
  int bytesin, bytesout;   pid_t childpid;
  int fd[2];

  pipe(fd);
  bytesin = strlen(bufin);
  childpid = Fork();
  if (childpid  != 0) {            /* parent */
    close(fd[0]);
    bytesout = write(fd[1], bufout, strlen(bufout)+1);
    printf("[%d]: wrote %d bytes\n", getpid(), bytesout);
  } else {                         /* child */
    close(fd[1]);
    bytesin = read(fd[0], bufin, BUFSIZE);
    printf("[%d]: read %d bytes, my bufin is {%s} \n »,
           getpid(), bytesin, bufin);
  }
  exit(0);
}
```

```
<unix>./parentwritepipe
[29196]:wrote 6 bytes
[29197]: read 6 bytes, my bufin is {hello}
<unix>
```

Unix pipes
Additional details

- Pipes are unidirectional (i.e., one-way communication), with first-in-first-out semantics
  - If two-way communication is needed, use a pair of pipes

- Pipes are not persistent

- **Automatic producer-consumer synchronization**
  - A reader will block if the pipe is empty but has at least one writer (i.e., the pipe input is still open)
  - If the pipe is empty and has no remaining writer, **`read`** will return 0
  - A writer will block if pipe is full but has at least one reader (i.e., the pipe output is still open)
  - A **`write`** to a pipe with a closed output will trigger an error
  - So, for correct operation, it is important for each process to close the unused side(s) of a given pipe

Unix pipes
Additional details (continued)

- A call to write on a pipe with less than **`PIPE_BUF`** bytes (4096 bytes on Linux) is an atomic operation
- A call to write on a pipe with more than **`PIPE_BUF`** bytes is not necessarily atomic (i.e., the written data may get interleaved with the data of other writes)
- **`lseek`** does not work on pipes
- See **`man 7 pipe`** for details

Named pipes (a.k.a. FIFOs)

- As we have seen, “basic” pipes can only be used between processes of the same family
- Named pipes (called “FIFOs”) remove this restriction

- A FIFO is created via the **`mkfifo`** system call and appears in the file system hierarchy
  - (and has corresponding access rights, like a regular file)

- A reader process must **`open`** the FIFO in read-only mode (**`O_RDONLY`**)
- A writer process must **`open`** the FIFO in write-only mode (**`O_WRONLY`**)
- **Rendez-vous between producer and consumer**: the first process that calls **`open`** is blocked; gets unblocked when the second process calls **`open`**

- A FIFO is persistent is the file system but the corresponding data buffer is not
- See **`man 7 pipe`** and **`man 7 fifo`** for details

Outline

- Introduction
- Basic Unix I/O interface
  - Main primitives
  - Kernel management of open files
- Unix standard I/O interface
- Inter-process communication via pipes and FIFOs
- Dealing with short counts – an example : the RIO library
- Wrap-up on Unix I/O interfaces

Repeated slide: Reading files

- Reading a file copies bytes from the current file position to memory, and then updates file position

```
char buf[512];
int fd;        /* file descriptor */
int nbytes;    /* number of bytes read */

/* Open file fd ...  */
/* Then read up to 512 bytes from file fd */
if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {
   perror("read");
   exit(1);
}
```

- Returns number of bytes read from file **`fd`** into **`buf`**
  - Return type **`ssize_t`** is signed integer (unlike **`size_t`**)
  - **`nbytes < 0`** indicates that an error occurred
  - ***Short counts*** (**`nbytes < sizeof(buf)`**) are possible and are not errors!

Dealing with short counts

- Short counts can occur in these situations:
  - Encountering end-of-file (EOF) on reads
  - Reading text lines from a terminal
  - Reading and writing network sockets or Unix pipes

- Short counts never occur in these situations:
  - Reading from disk files (except for EOF)
  - Writing to disk files

- One way to deal with short counts in your code:
  - Use the RIO (Robust I/O) package from the "CSAPP" textbook (see http://csapp.cs.cmu.edu)
  - The RIO functions are part of the **`csapp.h`** and **`csapp.c`** files available from: http://csapp.cs.cmu.edu/public/code.html
  - The RIO functions are explained in the following chapter: http://csapp.cs.cmu.edu/public/ch10-preview.pdf

The RIO package

- RIO is a set of wrappers that provide efficient and robust I/O in apps, such as network programs that are subject to short counts

- RIO provides two different kinds of functions
  - **Unbuffered input and output of binary data**
    - `rio_readn` and `rio_writen`
  - **Buffered input of binary data and text lines**
    - `rio_readlineb` and `rio_readnb`
    - Buffered RIO routines are ***thread-safe*** and can be interleaved arbitrarily on the same descriptor

- Note: **this is not a standard C/Unix package**
  - You should manually download the `csapp.h` and `csapp.c` files (see previous slide for details)

Unbuffered RIO

- Same interface as Unix **read** and **write**
- Especially useful for transferring data on pipes/network sockets

```
#include "csapp.h"

ssize_t rio_readn(int fd, void *usrbuf, size_t n);
ssize_t rio_writen(int fd, void *usrbuf, size_t n);
```

Return: num. bytes transferred if OK, 0 on EOF (rio_readn only), -1 on error

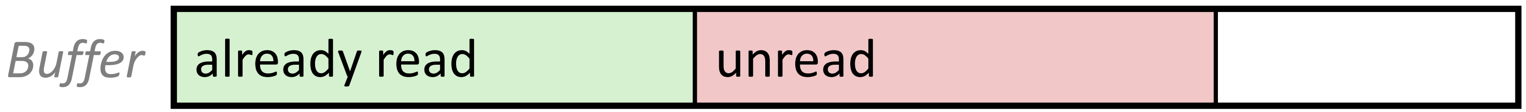
- **rio_readn** returns short count only if it encounters EOF
  - Only use it when you know how many bytes to read
- **rio_writen** never returns a short count
- Calls to **rio_readn** and **rio_writen** can be interleaved arbitrarily on the same descriptor

Implementation of `rio_readn`

```
/*
 * rio_readn - robustly read n bytes (unbuffered)
 */
ssize_t rio_readn(int fd, void *usrbuf, size_t n)
{
    size_t nleft = n;
    ssize_t nread;
    char *bufp = usrbuf;

    while (nleft > 0) {
        if ((nread = read(fd, bufp, nleft)) < 0) {
            if (errno == EINTR) /* interrupted by sig handler return */
                nread = 0;      /* and call read() again */
            else
                return -1;      /* errno set by read() */
        }
        else if (nread == 0)
            break;              /* EOF */
        nleft -= nread;
        bufp += nread;
    }
    return (n - nleft);         /* return >= 0 */
}
```

Buffered I/O: motivation

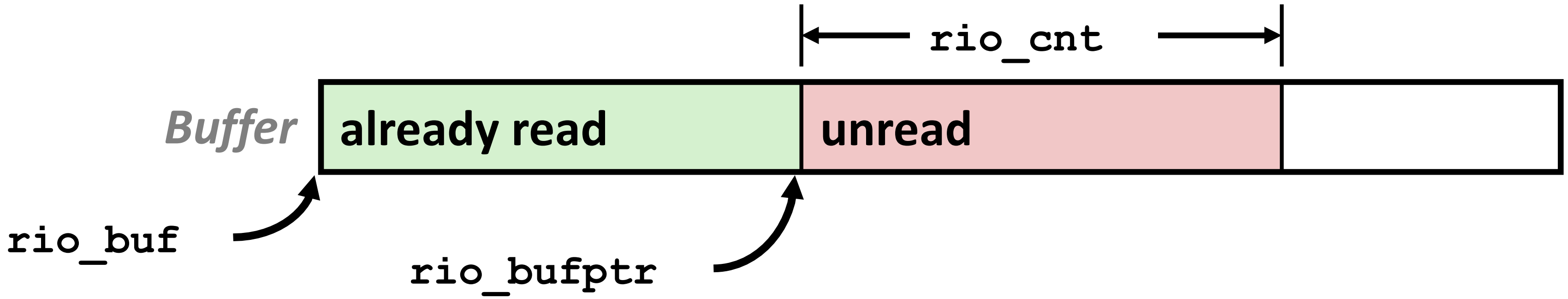
- I/O applications read/write one or a few characters at a time
  - **`getc, putc, ungetc`**
  - **`gets`**
    - Read line of text, stopping at newline or string delimiter
  - **`fscanf`**
- Implementing as calls to Unix I/O expensive
  - Read & Write involve Unix kernel calls
    - > 10,000 clock cycles



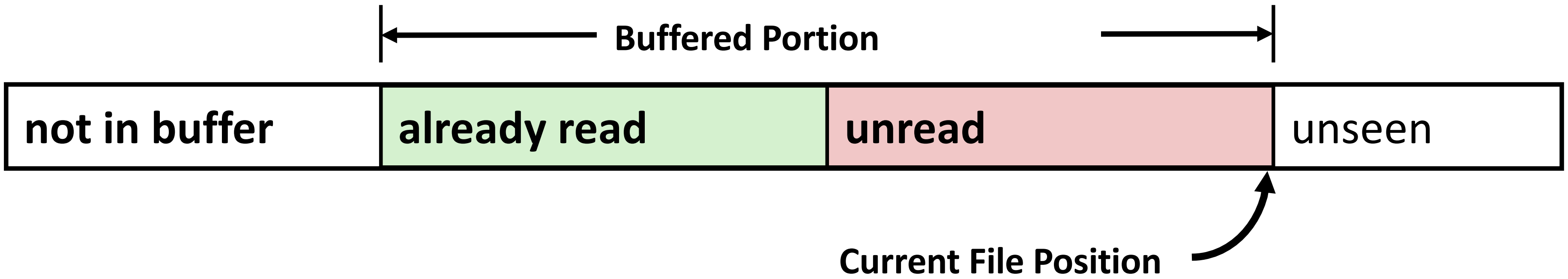
- Buffered read
  - Use Unix **`read()`** to grab block of bytes
  - User input functions take one (or a few) byte(s) at a time from buffer
    - Refill buffer when empty

Buffered I/O: implementation

- For reading from file
- File has associated buffer to hold bytes that have been read from file but not yet read by user code

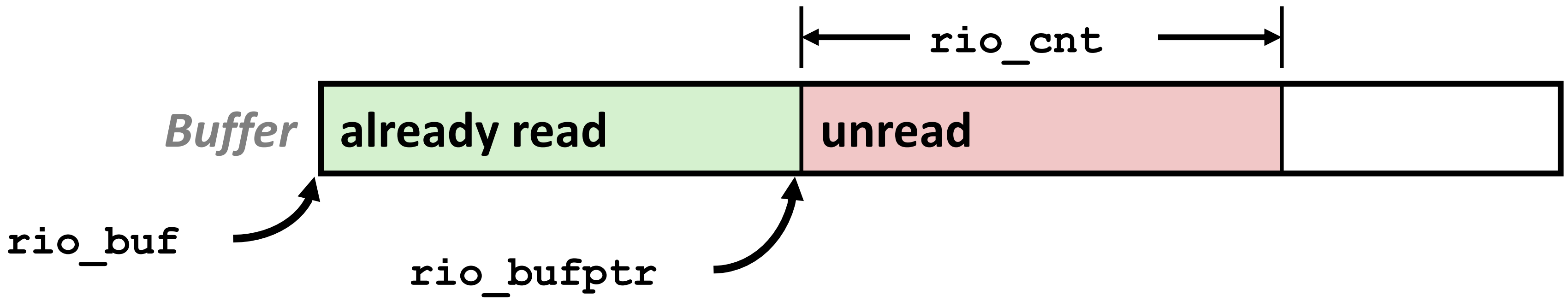


- Layered on Unix File



Buffered I/O: declaration

- All information contained in struct




```
typedef struct {
    int rio_fd;                /* descriptor for this internal buf */
    int rio_cnt;               /* unread bytes in internal buf */
    char *rio_bufptr;          /* next unread byte in internal buf */
    char rio_buf[RIO_BUFSIZE]; /* internal buffer */
} rio_t;
```

Buffered RIO input functions

- Efficiently read text lines and binary data from a file partially cached in an internal memory buffer

```
#include "csapp.h"

void rio_readinitb(rio_t *rp, int fd);

ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen);
```

Return: num. bytes read if OK, 0 on EOF, -1 on error

  - **`rio_readlineb`** reads a text line of up to **`maxlen`** bytes from file **`fd`** and stores the line in **`usrbuf`**
    - Especially useful for reading text lines from pipes/network sockets
  - Stopping conditions
    - **`maxlen`** bytes read
    - EOF encountered
    - Newline (‘**`\n`**’) encountered

Buffered RIO input functions (continued)

```
#include "csapp.h"

void rio_readinitb(rio_t *rp, int fd);

ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen);
ssize_t rio_readnb(rio_t *rp, void *usrbuf, size_t n);
```

Return: num. bytes read if OK, 0 on EOF, -1 on error

- **`rio_readnb`** reads up to **`n`** bytes from file **`fd`**
- Stopping conditions
  - **`maxlen`** bytes read
  - EOF encountered
- Calls to **`rio_readlineb`** and **`rio_readnb`** can be interleaved arbitrarily on the same descriptor
  - Warning: **<u>Do not interleave</u>** with calls to **`rio_readn`**

RIO example

- Copying the lines of a text file from standard input to standard output

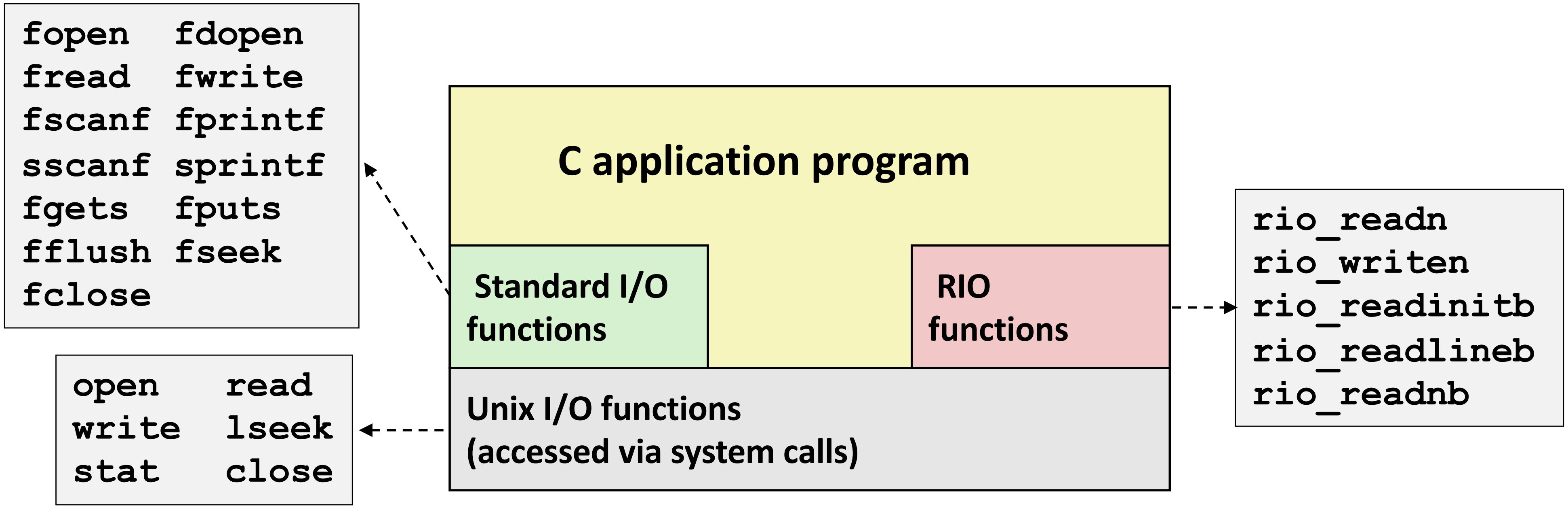
```
#include "csapp.h"

int main(int argc, char **argv)
{
    int n;
    rio_t rio;
    char buf[MAXLINE];

    Rio_readinitb(&rio, STDIN_FILENO);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0)
        Rio_writen(STDOUT_FILENO, buf, n);
    exit(0);
}
```

Outline

- Introduction
- Basic Unix I/O interface
  - Main primitives
  - Kernel management of open files
- Unix standard I/O interface
- Inter-process communication via pipes and FIFOs
- Dealing with short counts – an example : the RIO library
- Wrap-up on Unix I/O interfaces

Unix I/O vs. standard I/O vs. RIO

- Standard I/O and RIO are implemented using low-level Unix I/O



- Which ones should you use in your programs?

Choosing I/O functions

- **General rule: use the highest-level I/O functions you can**
  - Many C programmers are able to do all of their work using the standard I/O functions

- **When to use standard I/O** (**`fopen`**, **`fread`**, **`fwrite`** ...)
  - When working with disk or terminal files
- **When to use raw Unix I/O** (**`open`**, **`read`**, **`write`** ...)
  - When you need to fetch file metadata
  - In rare cases when you need absolute highest performance
- **When to use RIO**
  - When you are reading and writing network sockets or pipes
  - <u>Never use standard I/O on sockets or pipes</u>

Pros and cons of raw Unix I/O

- Pros
  - Unix I/O is the most general and lowest overhead form of I/O.
    - All other I/O packages are implemented using Unix I/O functions.
  - Unix I/O provides functions for accessing file metadata.

- Cons
  - Dealing with short counts is tricky and error prone.
  - Efficient reading of text lines requires some form of buffering, also tricky and error prone.
  - Both of these issues are addressed by the standard I/O and RIO packages.

Pros and cons of standard I/O

- Pros:
  - Buffering increases efficiency by decreasing the number of **`read`** and **`write`** system calls
  - Short counts are handled automatically

- Cons:
  - Provides no function for accessing file metadata
  - Standard I/O is not appropriate for input and output on pipes and network sockets
  - There are poorly documented restrictions on streams that interact badly with restrictions on pipes/sockets

Working with binary files

- Binary File Examples
  - Object code (produced by compilers)
  - Images (JPEG, GIF, …)
  - Arbitrary byte values

- Functions you should **NOT** use with binary files
  - Line-oriented I/O
    - **`fgets, scanf, printf, rio_readlineb`**
    - Interpret byte value **`0x0A`** (‘**`\n`**’) as special
    - Use **`rio_readn`** or **`rio_readnb`** instead
  - String functions
    - **`strlen, strcpy`**
    - Interpret byte value 0 as special

For further information

- A very good reference:
  - W. Richard Stevens & Stephen A. Rago, ***Advanced Programming in the Unix Environment***, 2nd Edition, Addison Wesley, 2005 (or 3rd edition, 2013)
    - Updated from Stevens'1993 book

- Stevens was a very good technical writer
  - Produced authoritative books on:
    - Unix programming
    - TCP/IP (the protocol that makes the Internet work)
    - Unix network programming
    - Unix IPC programming
  - Died in 1999
    - But others have taken up his legacy

For further information (continued)

- See also:
  - M. Kerrisk. The Linux Programming Interface (*a Linux and UNIX system Programming Handbook*). No Starch Press, 2010.