# 15-213

*"The course that gives CMU its Zip!"*

# System-Level I/O
# October 9, 2008

**Topics**

- Unix I/O
- Robust reading and writing
- Reading file metadata
- Sharing files
- I/O redirection
- Standard I/O

`lecture-13.ppt`

# Announcements

**Final exam day/time announced (by CMU)**

- **8:30-11:30am on Friday, December 12**

**Cheating… please, please don't**

- **Writing code together counts as "sharing code" – forbidden**
- **"Pair programming", even w/o looking at other's code – forbidden**
  - **describing code line by line counts the same as sharing code**
- **Opening up code and then leaving it for someone to enjoy – forbidden**
  - **in fact, please remember to use protected directories and screen locking**
- **Talking through a problem can include pictures (not code) – ok**

- **The automated tools for discovering cheating are incredibly good**
  - **… please don't test them**
- **Everyone has been warned multiple times**
  - **cheating on the remaining labs will receive no mercy**

# Unix Files

**A Unix *file* is a sequence of *m* bytes:**

- $B_0, B_1, .... , B_k , .... , B_{m-1}$

**All I/O devices are represented as files:**

- `/dev/sda2` (`/usr` disk partition)
- `/dev/tty2` (terminal)

**Even the kernel is 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

## Socket

- A file type used for network comm. between processes

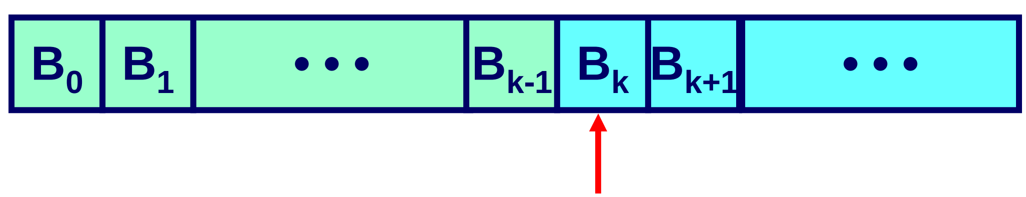
# Unix I/O

## 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 ()`




**Current File Position = k**

# 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 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
- `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 in to standard out, 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*

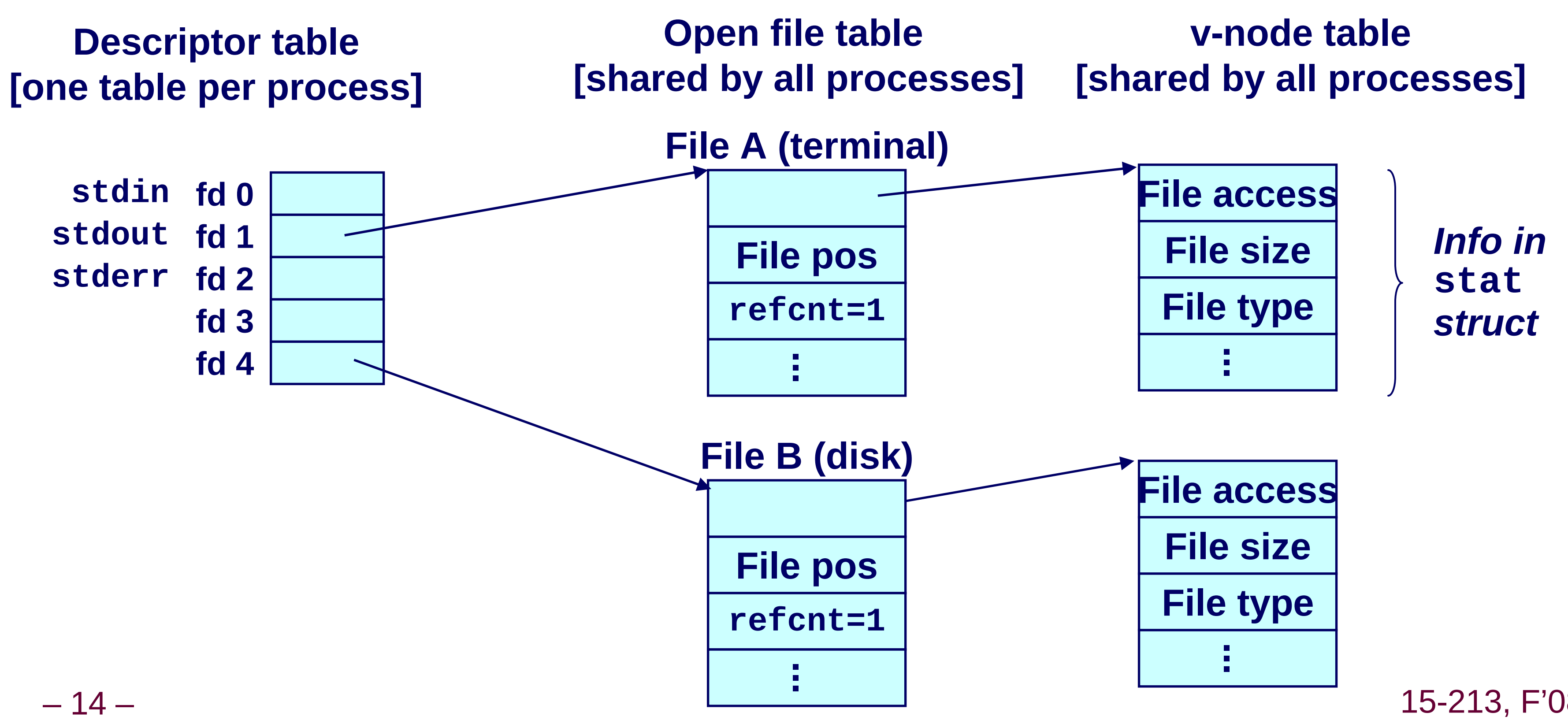
- `fd == -1` indicates that an error occurred

Each process created by a Unix shell begins life with three open files associated with a terminal:

- 0: standard input (stdin)
- 1: standard output (stdout)
- 2: standard error (stderr)

# How the Unix Kernel Represents Open Files

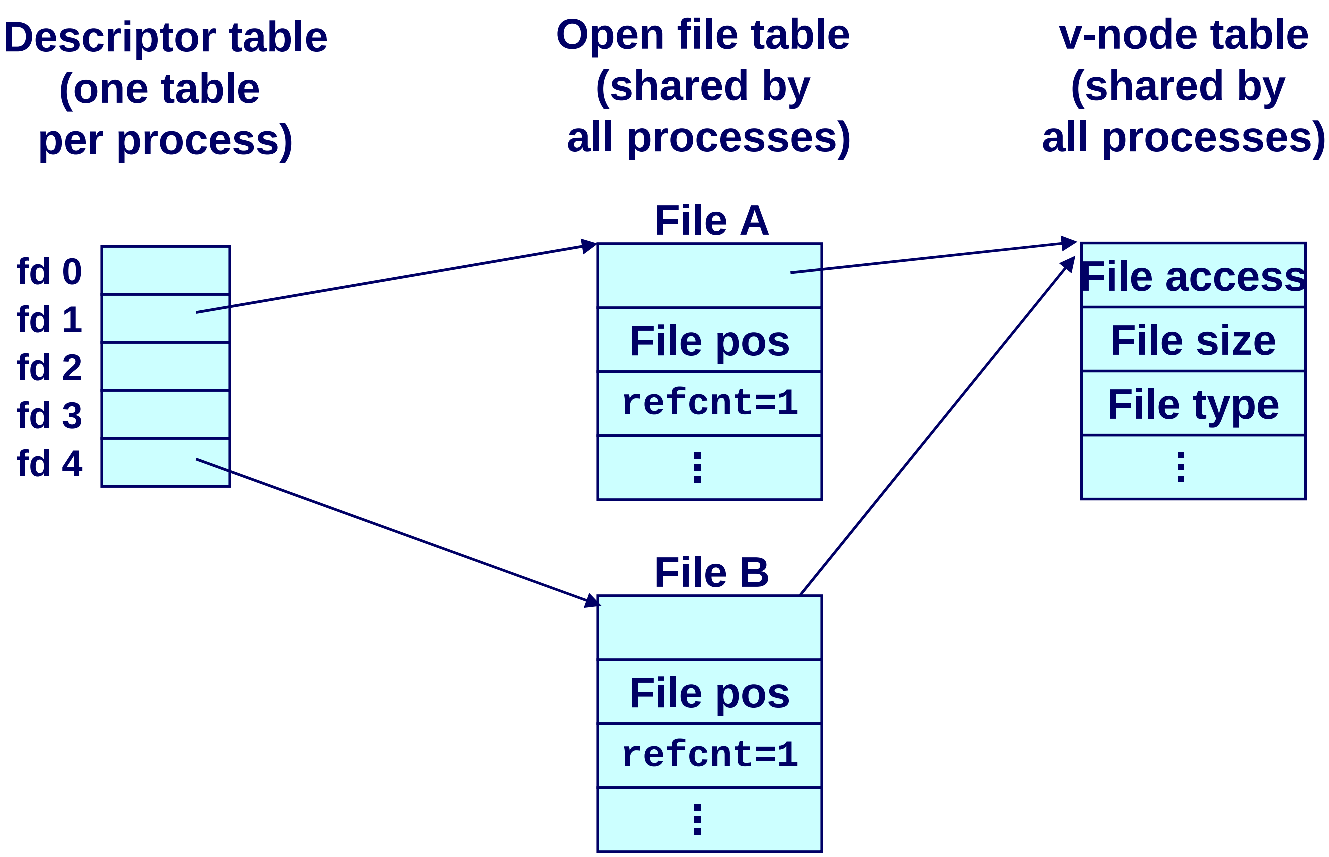
Two descriptors referencing two distinct open disk files. Descriptor 1 (stdout) points to terminal, and descriptor 4 points to open disk file



# 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

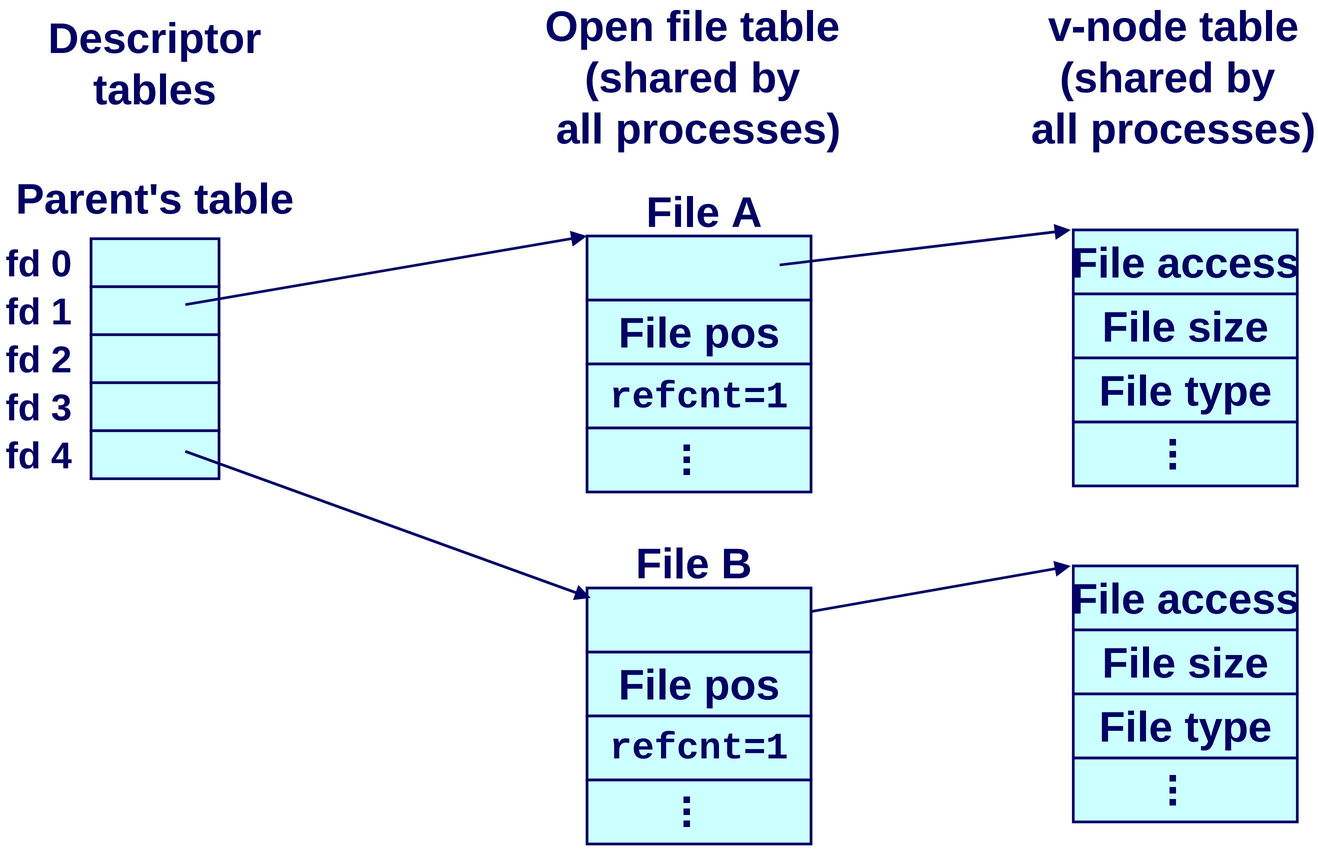


# How Processes Share Files

## A child process inherits its parent's open files

- `Note: situation unchanged by exec() functions`

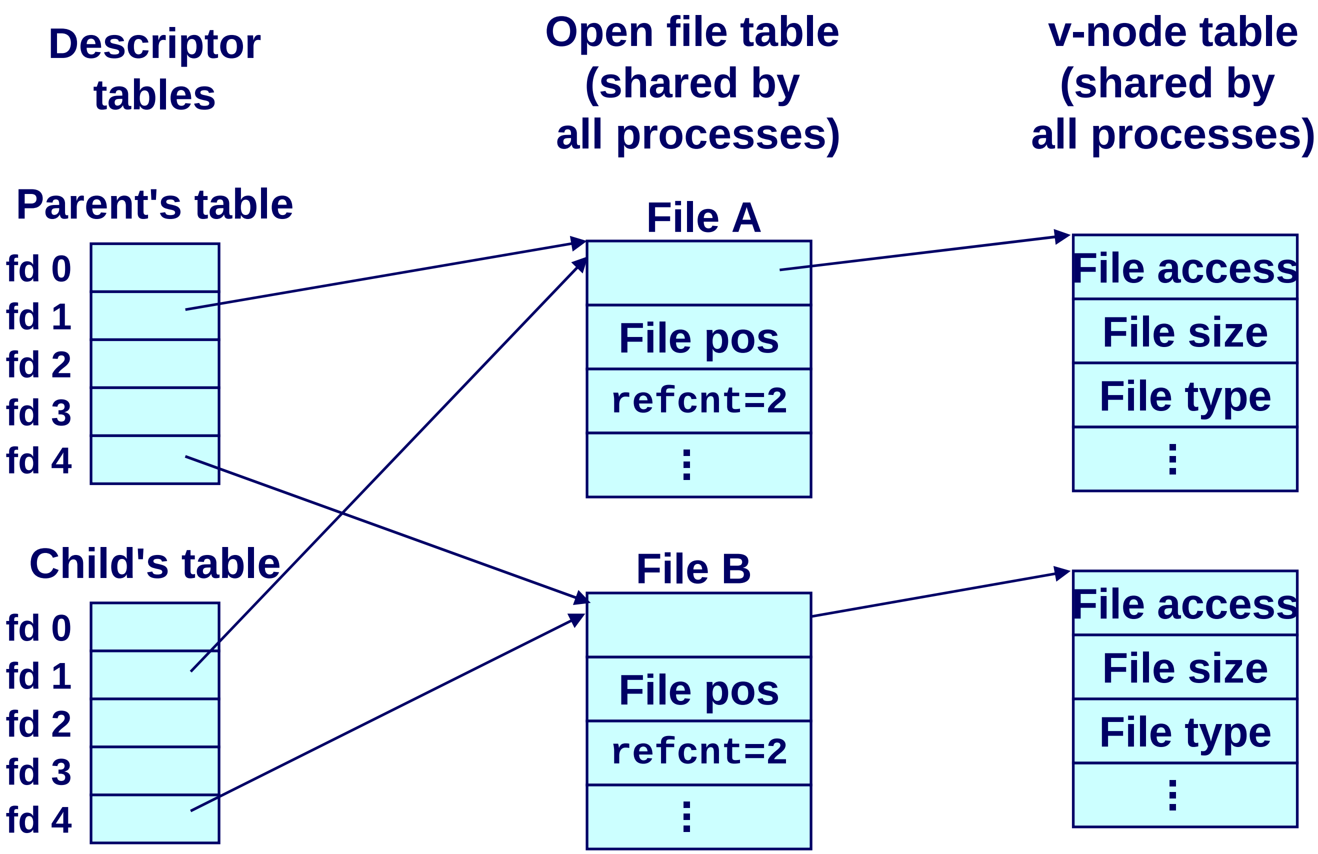
### Before fork() call



# Parent and child file sharing after fork()

## A child process inherits its parent's open files

- `Child's table same as parents, and +1 to each refcnt`
- `Note: situation unchanged by exec() functions`



# I/O Redirection

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

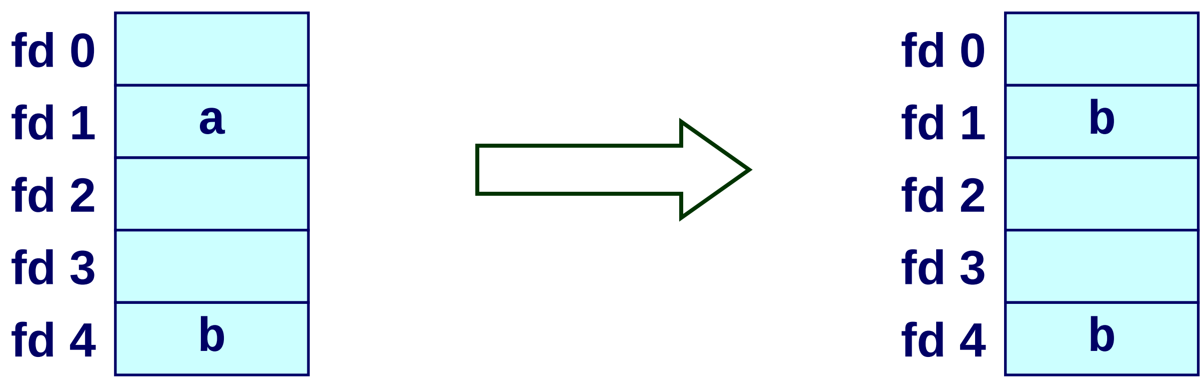
```
unix> ls > foo.txt
```

**Answer: By calling the `dup2(oldfd, newfd)` function**

- **Copies (per-process) descriptor table entry `oldfd` to entry `newfd`**

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

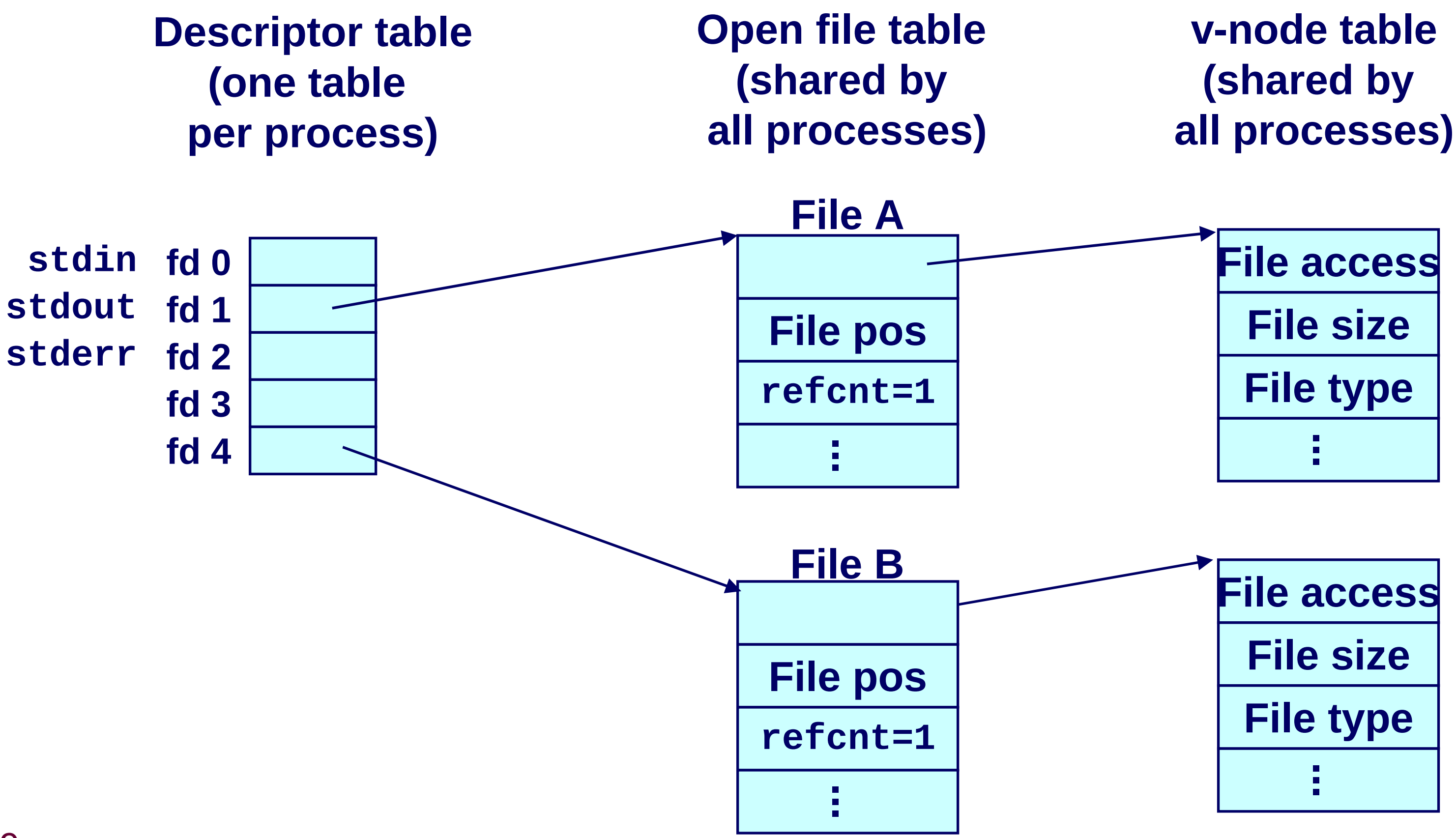
**Descriptor table after `dup2(4,1)`**



# I/O Redirection example

## Step #1: open file to which stdout should be redirected

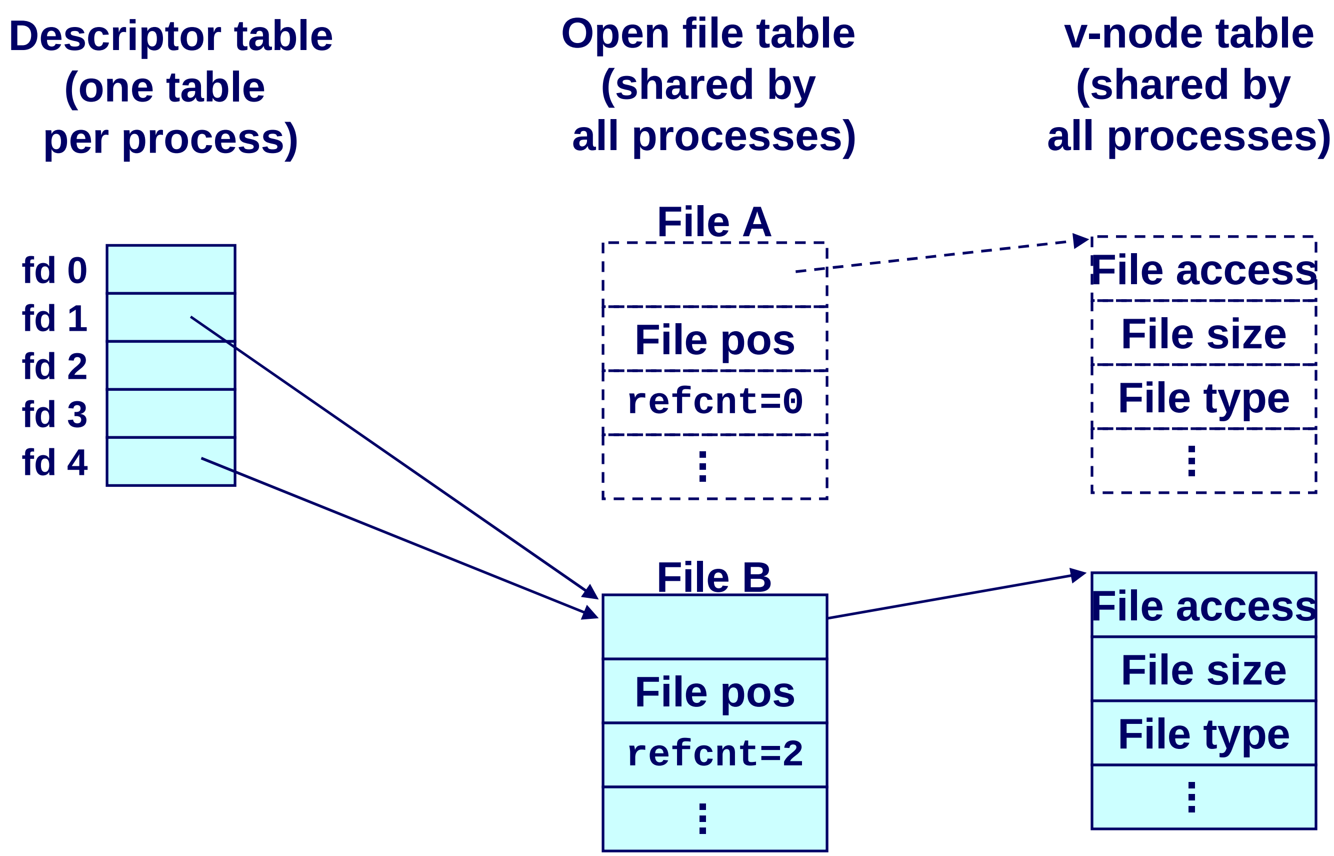
- Happens in child executing shell code, before exec()



# I/O Redirection Example (cont)

## Step #2: call `dup2(4,1)`

- cause fd=1 (stdout) to refer to disk file pointed at by fd=4



# Standard I/O Functions

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

- Documented in Appendix B of K&R.

Examples of standard I/O functions:

- 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 memory.
- Similar to buffered RIO

## C programs begin life with three open streams (defined in `stdio.h`)

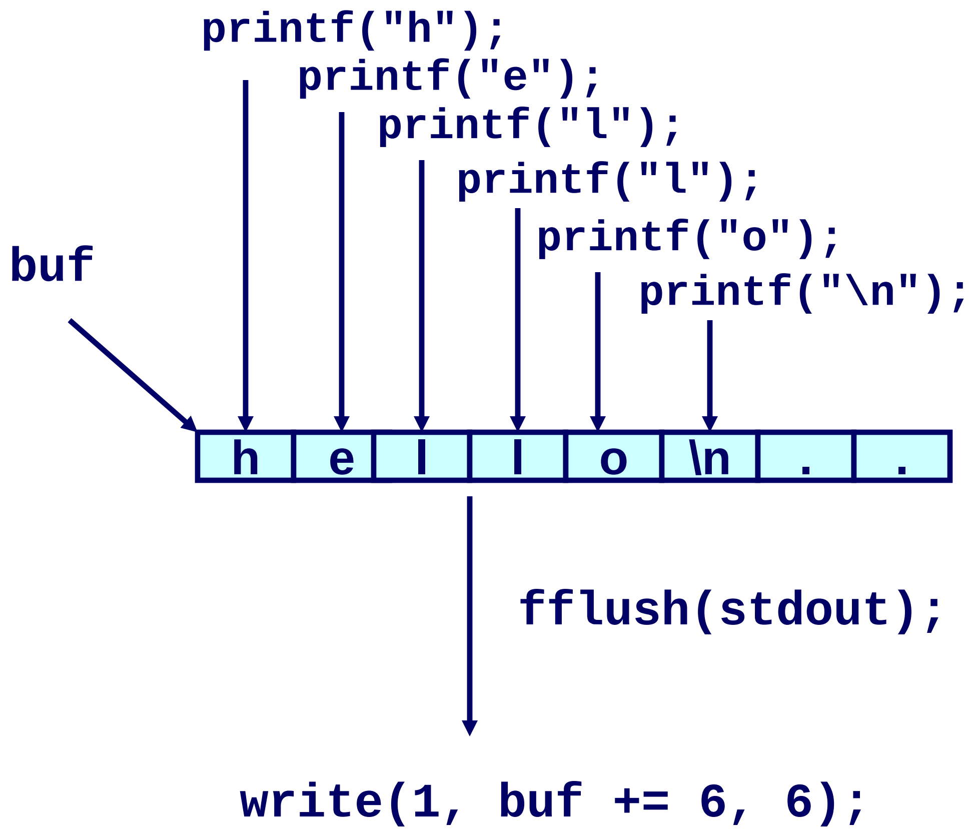
- `stdin` (standard input)
- `stdout` (standard output)
- `stderr` (standard error)

```
#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");
}
```

# 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 always fascinating Unix `strace` program:

```
#include <stdio.h>

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

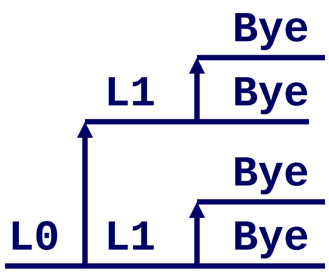
```
linux> strace ./hello
execve("./hello", ["hello"], [/* ... */]).
...
write(1, "hello\n", 6...)                  = 6
...
_exit(0)                                   = ?
```

# Fork Example #2 (Original)

## Key Points

- Both parent and child can continue forking

```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```





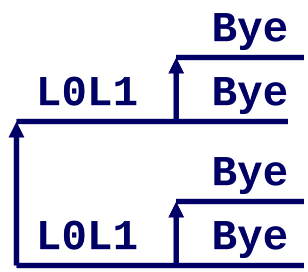
From lecture-13.ppt

# Fork Example #2 (modified)

## Removed the "\n" from the first printf

- As a result, "L0" gets printed twice

```
void fork2a()
{
    printf("L0");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



# 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**
- **`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 your textbook's `csapp.c` file (Appendix B)

# 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

Download from

```
csapp.cs.cmu.edu/public/ics/code/src/csapp.c
csapp.cs.cmu.edu/public/ics/code/include/csapp.h
```

# Unbuffered RIO Input and Output

**Same interface as Unix `read` and `write`**

**Especially useful for transferring data on 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 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

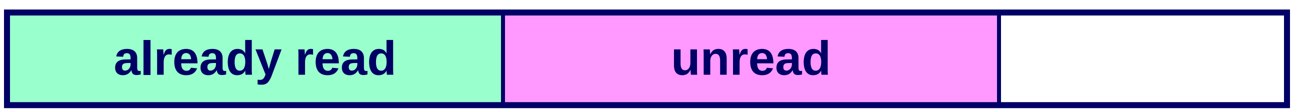
## I/O Applications Read/Write One Character at a Time

- getc, putc, ungetc
- gets
  - Read line of text, stopping at newline

## Implementing as Calls to Unix I/O Expensive

- Read & Write involve require Unix kernel calls
  - > 10,000 clock cycles

Buffer

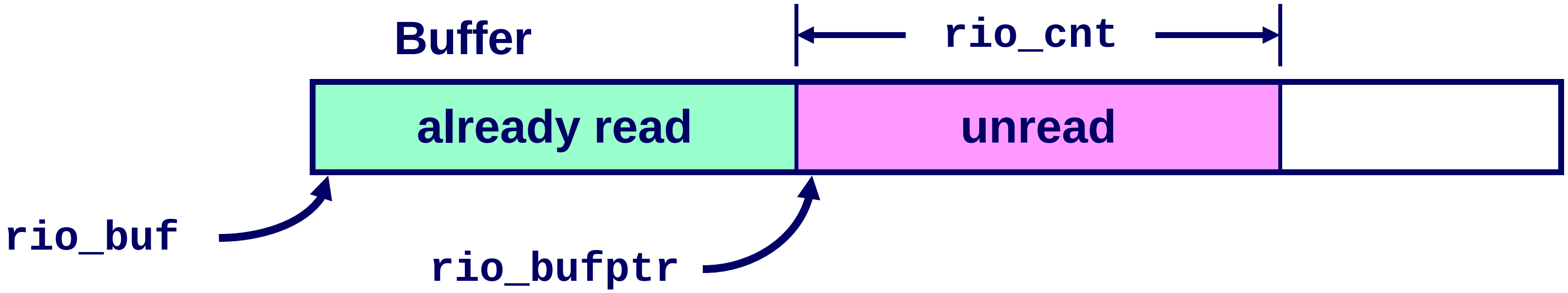



## Buffered Read

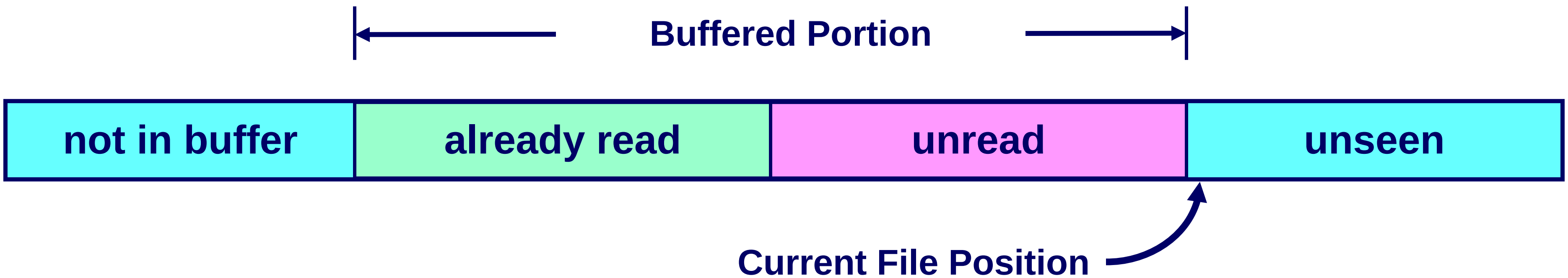
- Use Unix read to grab block of bytes
- User input functions take one byte at a time from buffer
  - Refill buffer when empty

# 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

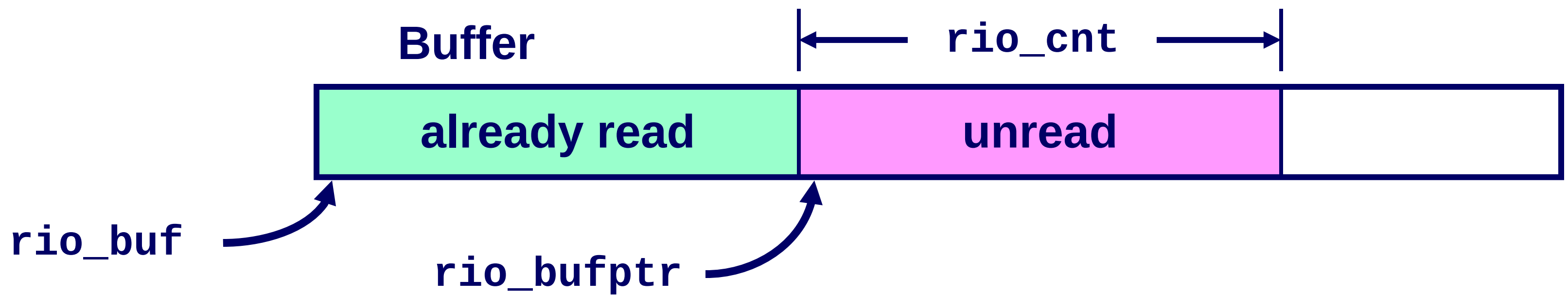



## Layered on Unix File



# 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 network sockets
- Stopping conditions
  - `maxlen` bytes read
  - EOF encountered
  - Newline (‘`\n`’) encountered

# Buffered RIO Input Functions (cont)

```
#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: Don't interleave 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);
}
```

# 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

- When working with disk or terminal files

## When to use raw Unix I/O

- 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
- Never use standard I/O or raw Unix I/O on sockets or pipes

# For Further Information

## The Unix bible:

- W. Richard Stevens & Stephen A. Rago, Advanced Programming in the Unix Environment, 2nd Edition, Addison Wesley, 2005
  - Updated from Stevens' 1993 book

## Stevens is arguably the best technical writer ever.

- Produced authoritative works in:
  - Unix programming
  - TCP/IP (the protocol that makes the Internet work)
  - Unix network programming
  - Unix IPC programming

## Tragically, Stevens died Sept 1, 1999

- But others have taken up his legacy

# Fun with File Descriptors (1)

```
#include "csapp.h"
int main(int argc, char *argv[])
{
    int fd1, fd2, fd3;
    char c1, c2, c3;
    char *fname = argv[1];
    fd1 = Open(fname, O_RDONLY, 0);
    fd2 = Open(fname, O_RDONLY, 0);
    fd3 = Open(fname, O_RDONLY, 0);
    Dup2(fd2, fd3);
    Read(fd1, &c1, 1);
    Read(fd2, &c2, 1);
    Read(fd3, &c3, 1);
    printf("c1 = %c, c2 = %c, c3 = %c\n", c1, c2, c3);
    return 0;
}
```

- **What would this program print for file containing “abcde”?**

# Fun with File Descriptors (2)

```
#include "csapp.h"
int main(int argc, char *argv[])
{
    int fd1;
    int s = getpid() & 0x1;
    char c1, c2;
    char *fname = argv[1];
    fd1 = Open(fname, O_RDONLY, 0);
    Read(fd1, &c1, 1);
    if (fork()) {
        /* Parent */
        sleep(s);
        Read(fd1, &c2, 1);
        printf("Parent: c1 = %c, c2 = %c\n", c1, c2);
    } else {
        /* Child */
        sleep(1-s);
        Read(fd1, &c2, 1);
        printf("Child: c1 = %c, c2 = %c\n", c1, c2);
    }
    return 0;
}
```

- **What would this program print for file containing “abcde”?**

# Fun with File Descriptors (3)

```
#include "csapp.h"
int main(int argc, char *argv[])
{
    int fd1, fd2, fd3;
    char *fname = argv[1];
    fd1 = Open(fname, O_CREAT|O_TRUNC|O_RDWR, S_IRUSR|S_IWUSR);
    Write(fd1, "pqrs", 4);
    fd3 = Open(fname, O_APPEND|O_WRONLY, 0);
    Write(fd3, "jklmn", 5);
    fd2 = dup(fd1);  /* Allocates descriptor */
    Write(fd2, "wxyz", 4);
    Write(fd3, "ef", 2);
    return 0;
}
```

- **What would be contents of resulting file?**

# Accessing Directories

The only recommended operation on a directory is to read its entries

- dirent structure contains information about a directory entry
- DIR structure contains information about directory while stepping through its entries

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

{
  DIR *directory;
  struct dirent *de;
  ...
  if (!(directory = opendir(dir_name)))
      error("Failed to open directory");
  ...
  while (0 != (de = readdir(directory))) {
      printf("Found file: %s\n", de->d_name);
  }
  ...
  closedir(directory);
}
```

# Unix I/O Key Characteristics

## Classic Unix/Linux I/O:

### I/O operates on linear streams of Bytes

- Can reposition insertion point and extend file at end

### I/O tends to be synchronous

- Read or write operation block until data has been transferred

### Fine grained I/O

- One key-stroke at a time
- Each I/O event is handled by the kernel and an appropriate process

## Mainframe I/O:

### I/O operates on structured records

- Functions to locate, insert, remove, update records

### I/O tends to be asynchronous

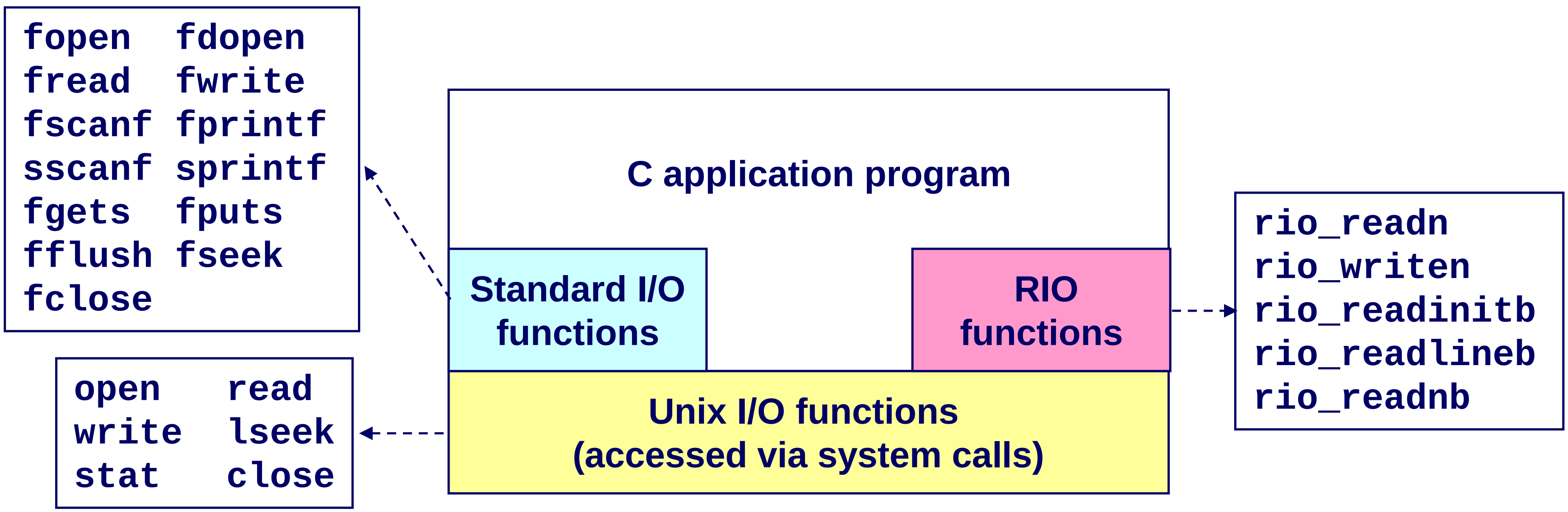
- Overlap I/O and computation within a process

### Coarse grained I/O

- Process writes “channel programs” to be executed by the I/O hardware
- Many I/O operations are performed autonomously with one interrupt at completion

# 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?

# Pros and Cons of 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 network sockets
- There are poorly documented restrictions on streams that interact badly with restrictions on sockets

# Working with Binary Files

## Binary File Examples

- Object code
- Images (JPEG, GIF)
- Arbitrary byte values

## Functions you shouldn't use

- Line-oriented I/O
  - fgets, scanf, printf, rio_readlineb
    - » use rio_readn or rio_readnb instead
  - Interprets byte value `0x0A` ('`\n`') as special
- String functions
  - strlen, strcpy
  - Interprets byte value 0 as special

# Java I/O

## Standard Java Streams are Unbuffered

- Every read/write call invokes OS
- Preferable to "wrap" stream with buffered stream

## Java Distinguishes Characters from Bytes

- Characters: Various encodings to allow more than ASCII characters

```
BufferedReader in =
  new BufferedReader(new FileReader("char-in.txt"));
BufferedWriter out =
  new BufferedWriter(new FileWriter("char-out.txt"));
```

- Bytes: Always 8 bits.  Used for binary data

```
BufferedInputStream in =
  new BufferedInputStream(new FileInputStream("binary-in.txt"));
BufferedOutputStream out =
  new BufferedOutputStream(new FileOutputStream("binary-out.txt"));
```