Systems Programming

System-Level I/O

Byoungyoung Lee

Seoul National University

byoungyoung@snu.ac.kr

https://lifeasageek.github.io

Lecture slides are prepared based on materials provided by CSAPP authors.

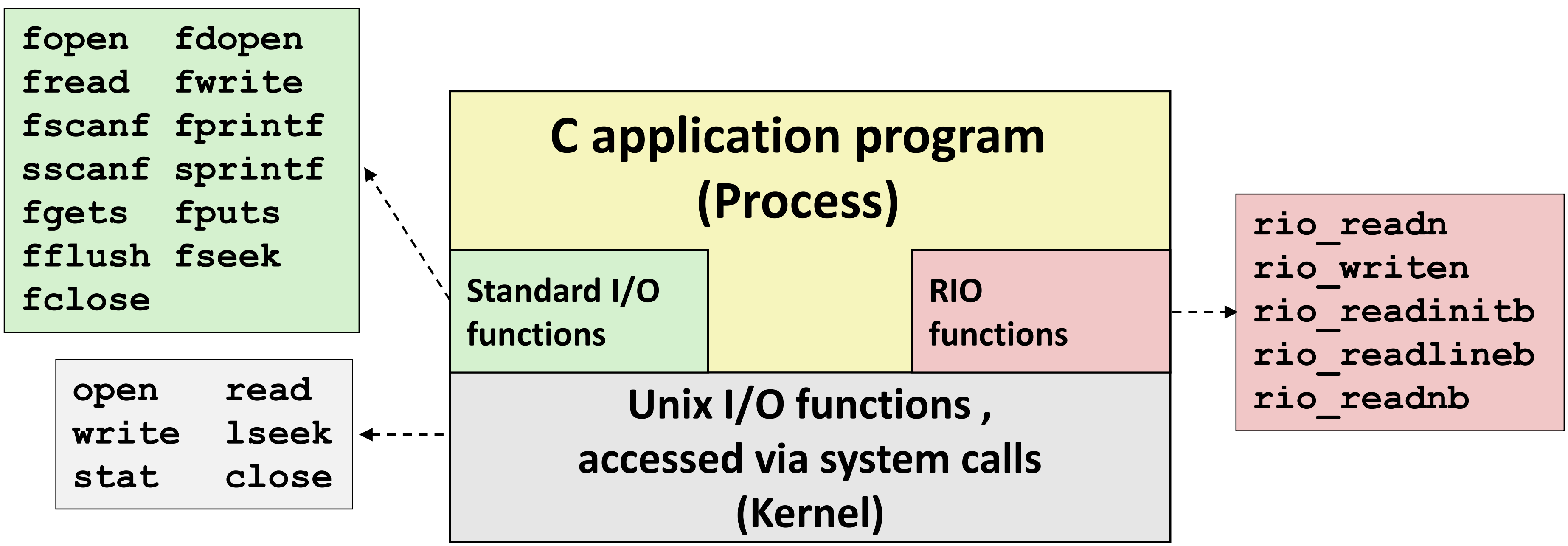
Today

- **Unix I/O**
- Metadata, sharing, and redirection
- Standard I/O
- Closing remarks

Today: Unix I/O and C Standard I/O

- **Two sets: system-level and C-level**
- **Robust I/O (RIO)**
  - 213 special wrappers
  - good coding practice: handles error checking, signals, and "short counts"

  ➔ Check the textbook!

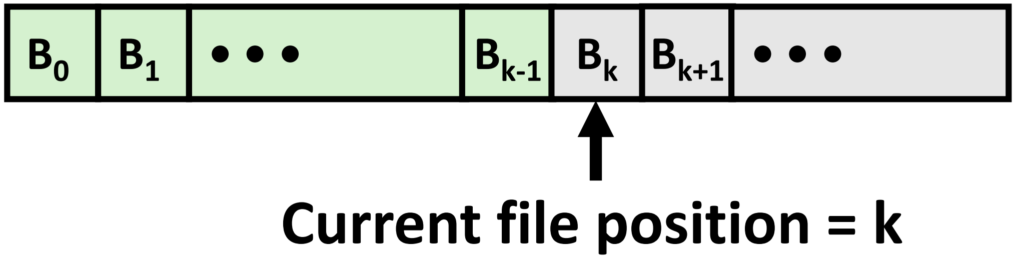


Unix I/O Overview

- **A Linux *file* is a sequence of *m* bytes:**
  - $B_0, B_1, \ldots, B_k, \ldots, B_{m-1}$
- **Almost everything is a file in Linux**
  - All I/O devices are represented as files:
    - **`/dev/sda2`** (e.g., **`/usr`** disk partition)
    - **`/dev/tty2`** (terminal)
  - The kernel is represented as a file:
    - **`/boot/vmlinuz-3.13.0-55-generic`** (kernel image)
    - **`/proc`** (pseudo files to access kernel data structures)

Unix I/O Overview

- **Elegant mapping of files to devices allows kernel to export simple interface called *Unix I/O:***
  - Opening and closing files
    - **`open()`** and **`close()`**
  - Reading and writing a file
    - **`read()`** and **`write()`**
  - Changing the ***current file position*** (seek)
    - indicates an offset to read from or write to
    - **`lseek()`**



File Types

- **Each file has a *type* indicating its role in the system**
  - *Regular file:* Contains arbitrary data
  - *Directory:* Index for a related group of files
  - *Socket:* For communicating with a process on another machine

- **Other file types beyond our scope**
  - *Named pipes (FIFOs)*
  - *Symbolic links*
  - *Character and block devices*

Regular Files

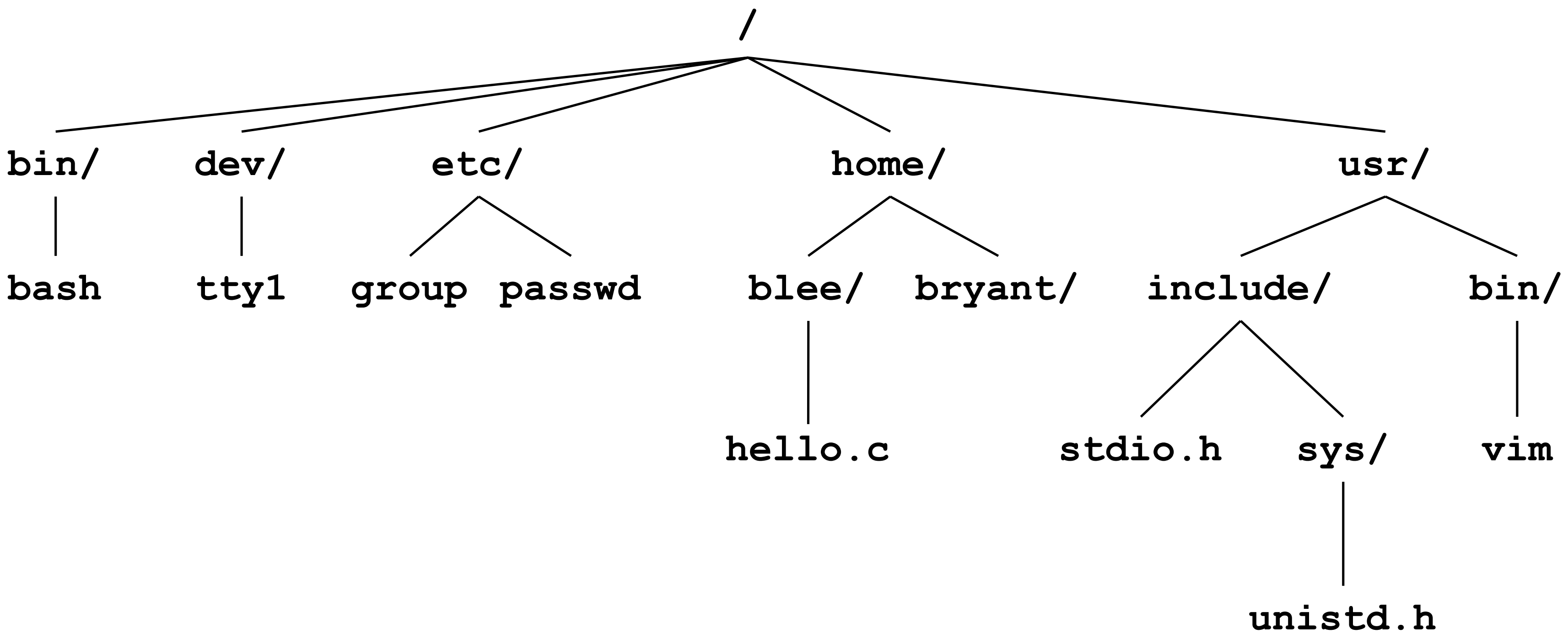
- **A regular file contains arbitrary data**
- **Applications often distinguish between *text files* and *binary files***
  - Text files are regular files with only ASCII or Unicode characters
  - Binary files are everything else
    - e.g., object files, JPEG images
  - Kernel doesn’t know the difference!
- **Text file is sequence of *text lines***
  - Text line is sequence of chars terminated by *newline char* (**‘`\n`’**)
- **End of line (EOL) indicators in other systems**
  - Linux and Mac OS: **‘`\n`’** (`0xa`)
    - line feed (LF)
  - Windows: **‘`\r\n`’** (`0xd 0xa`)
    - Carriage return (CR) followed by line feed (LF)

Directories

- **Directory consists of an array of *links***
  - Each array entry links to a file
- **Each directory contains at least two entries**
  - `.` (dot) is a link to itself
  - `..` (dot dot) is a link to *the parent directory* in the *directory hierarchy* (next slide)
- **Commands for manipulating directories**
  - **`mkdir`**: create empty directory
  - **`ls`**: view directory contents
  - **`rmdir`**: delete empty directory

Directory Hierarchy

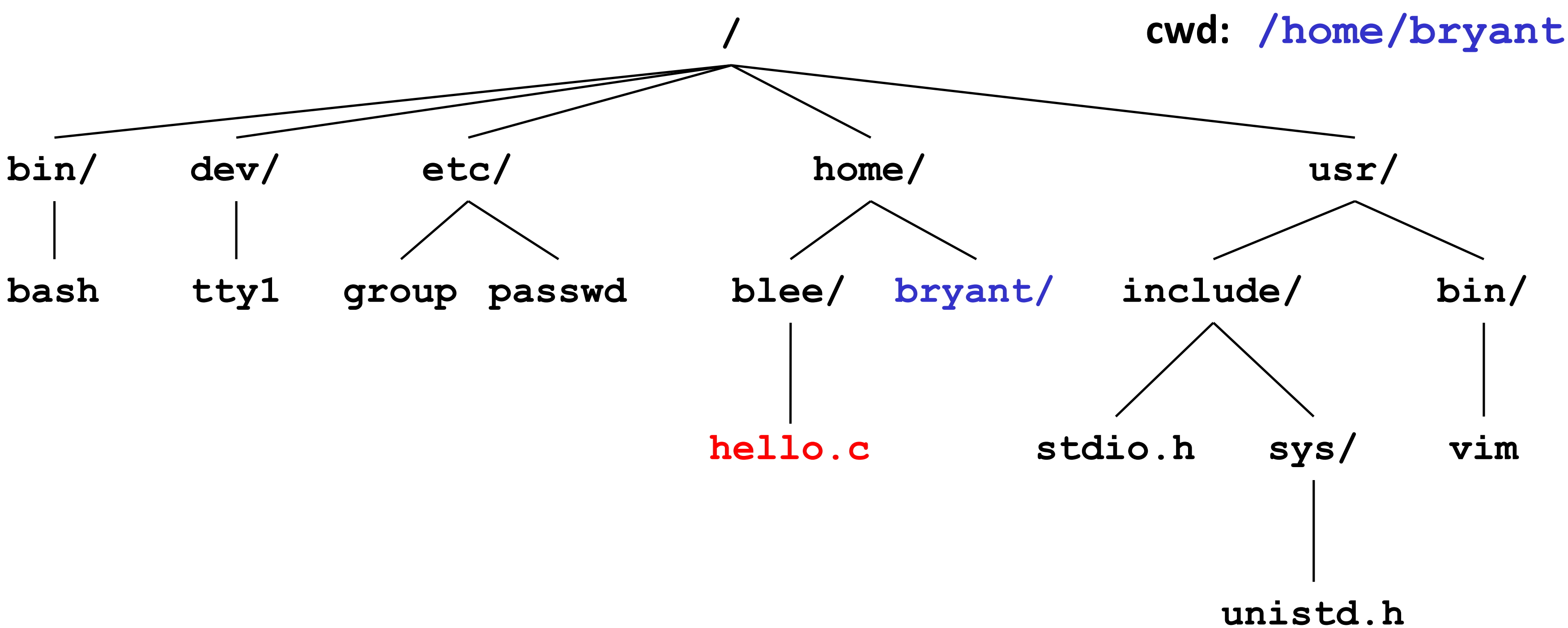
- **All files are organized as a hierarchy anchored by root directory named / (slash)**



- **Kernel maintains *current working directory (cwd)* for each process**
  - Modified using the **`cd`** command

Pathnames

- **Locations of files in the hierarchy denoted by *pathnames***
  - *Absolute pathname* starts with '/' and denotes path from root
    - **`/home/blee/hello.c`**
  - *Relative pathname* denotes path from current working directory
    - **`../blee/hello.c`**



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 begins life with three open files associated with a terminal:**
  - 0: standard input (stdin)
  - 1: standard output (stdout)
  - 2: standard error (stderr)

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**
  - Before close(), always check if fd is valid (i.e., fd != -1)
  - After close(), assign an invalid fd (i.e., fd = -1)

- **Q. When this double-close can be an issue?**
  - Compared to double-free?

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 (cf. `size_t` is unsigned 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 file to stdout, one byte at a time**

```
int main(int argc, char *argv[])
{
    char c;
    int infd = STDIN_FILENO;
    if (argc == 2) {
        infd = open(argv[1], O_RDONLY, 0);
    }
    while(read(infd, &c, 1) != 0)
        write(STDOUT_FILENO, &c, 1);
    exit(0);
}
```

showfile1_nobuf.c

- **Demo:**

```
$ strace ./showfile1_nobuf names.txt
```

On 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

- **Best practice is to always allow for short counts**

Home-Grown Buffered I/O Code

- **Copying file to stdout, BUFSIZE bytes at a time**

```
#define BUFSIZE 64


int main(int argc, char *argv[])
{
    char buf[BUFSIZE];
    int infd = STDIN_FILENO;
    if (argc == 2) {
        infd = open(argv[1], O_RDONLY, 0);
    }
    while((nread = read(infd, buf, BUFSIZE)) != 0)
        write(STDOUT_FILENO, buf, nread);
    exit(0);
}
```

showfile2_buf.c

- **Demo:**

```
$ strace ./showfile2_buf names.txt
```

Today

- Unix I/O
- **Metadata, sharing, and redirection**
- Standard I/O
- Closing remarks

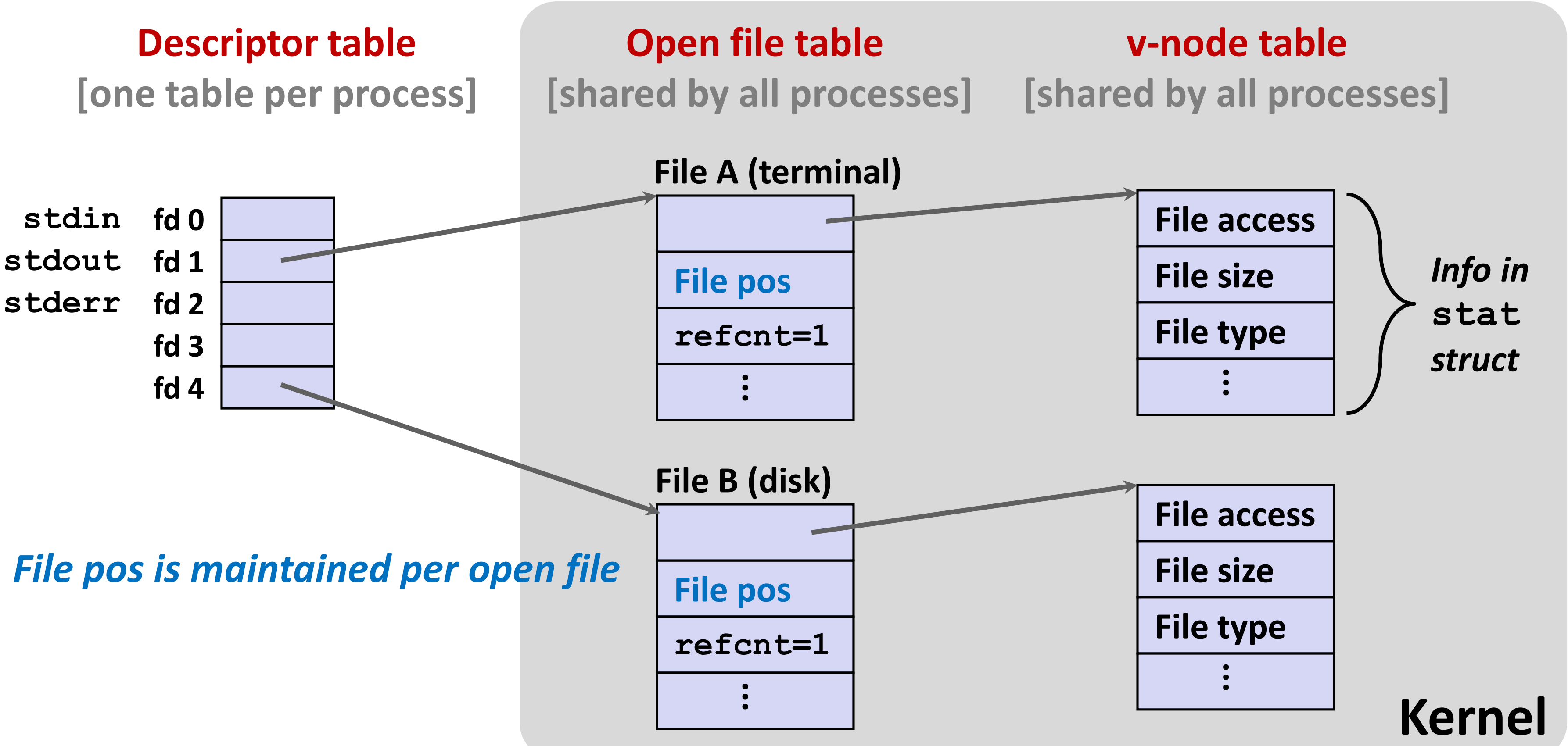
File Metadata

- ***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 */
};
```

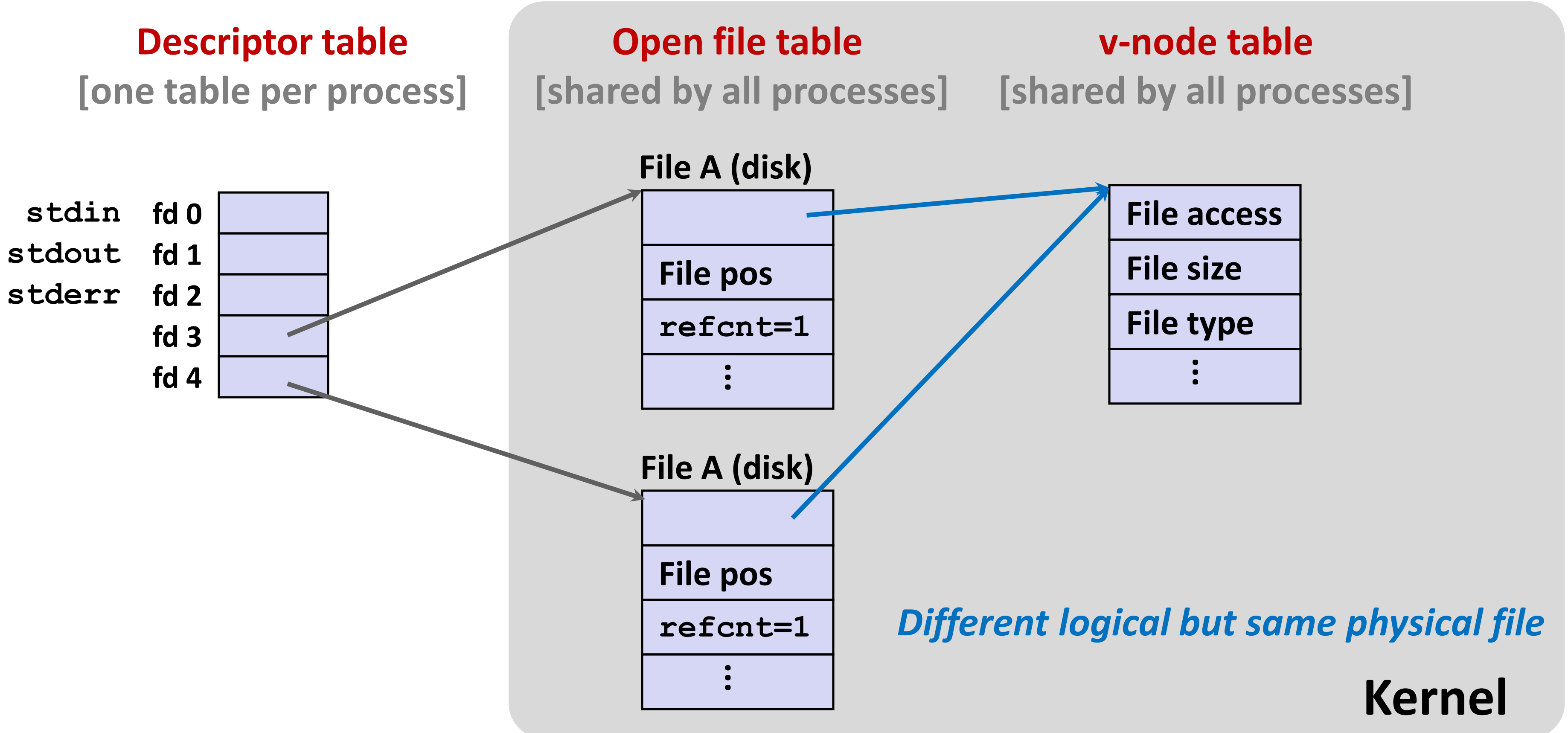
How the Unix Kernel Represents Open Files

- **Two descriptors referencing two distinct open files**
  - descriptor 1 (`stdout` or fd1) points to terminal
  - descriptor 4 (fd4) points to open a disk file



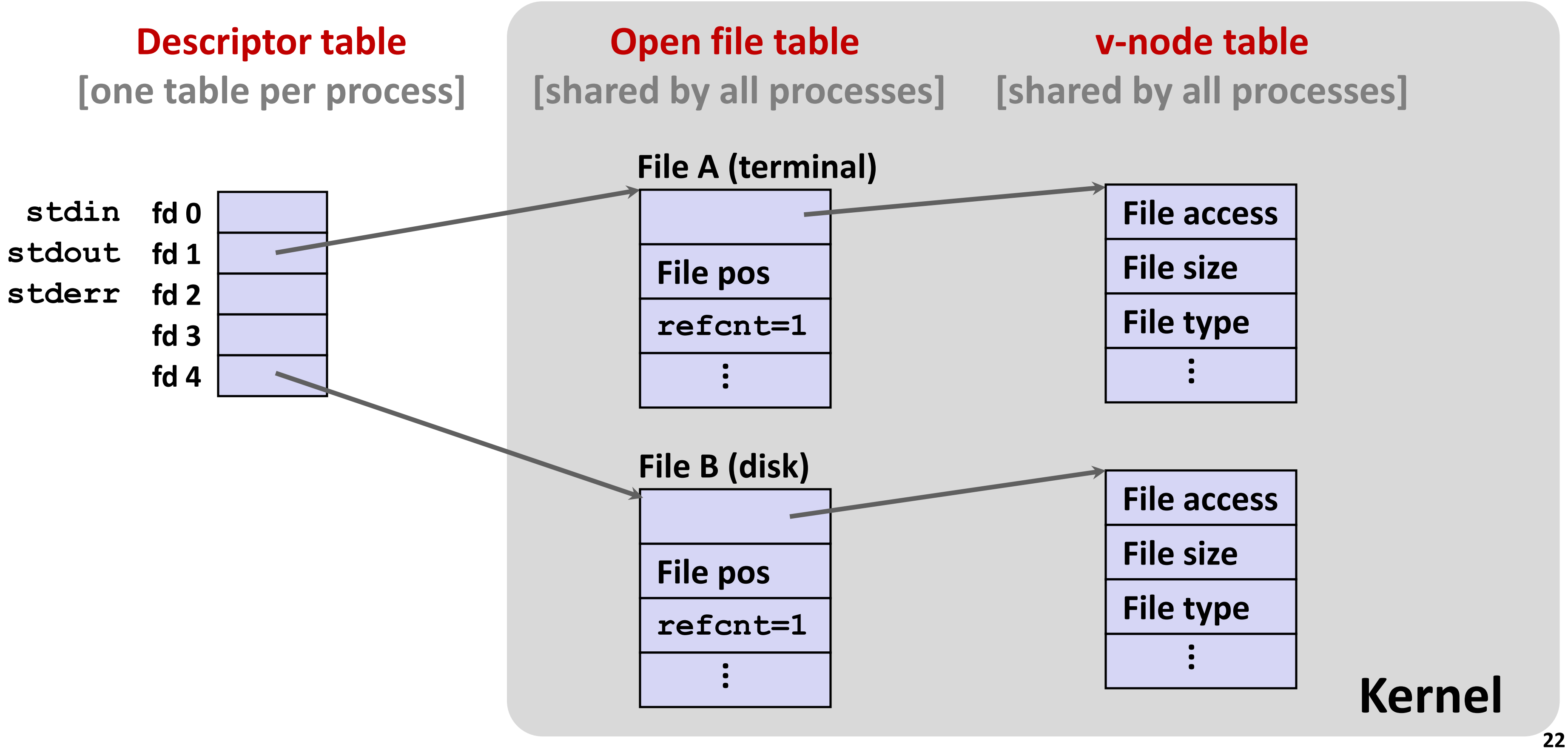
File Sharing

- **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



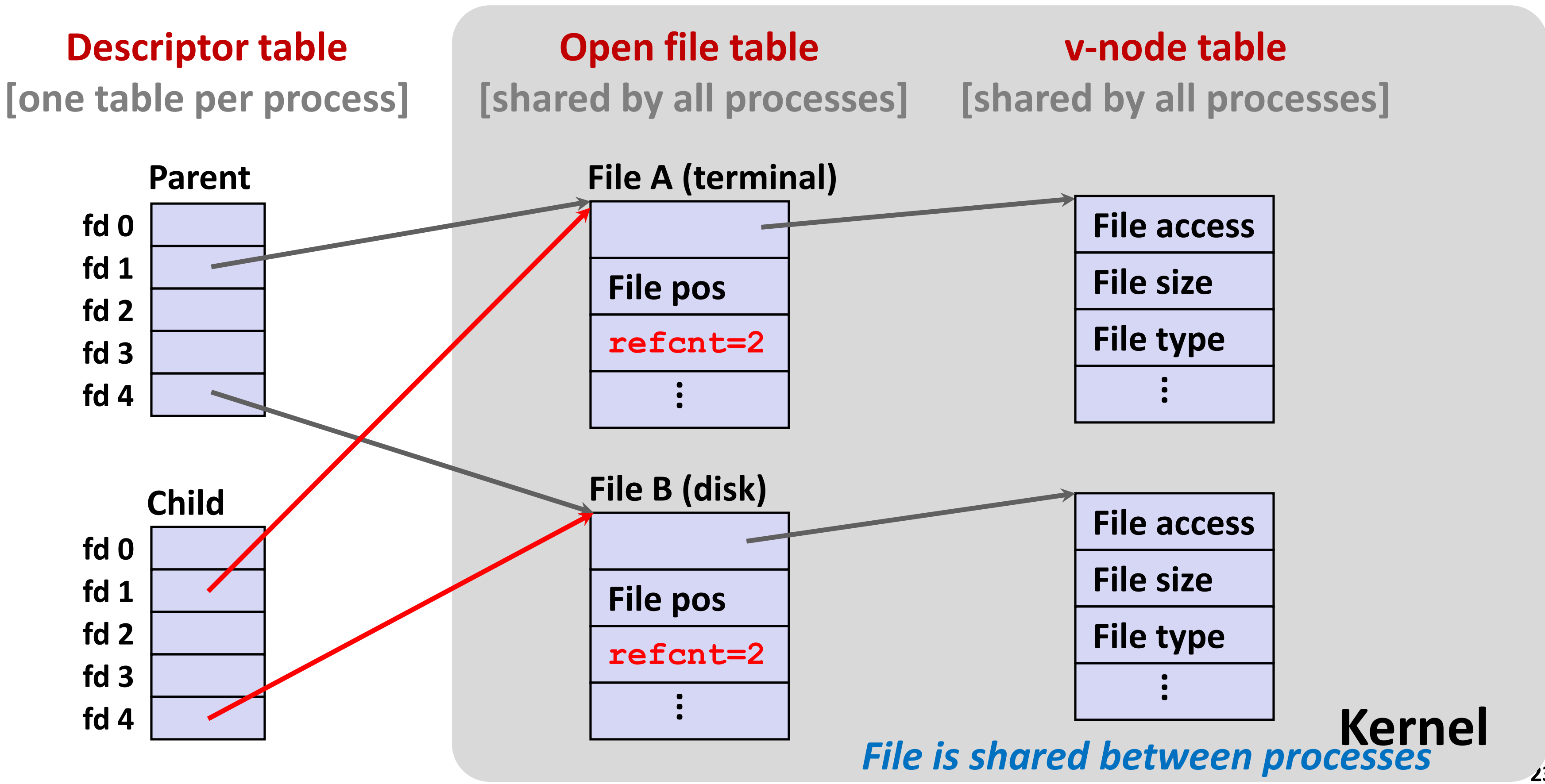
How Processes Share Files: `fork`

- **A child process inherits its parent's open files**
  - Note: situation unchanged by **`exec`** functions (use **`fcntl`** to change)
  - Check open syscall's O_CLOEXEC
- ***Before*** **`fork`** **call:**



How Processes Share Files: `fork`

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



I/O Redirection

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

```
$ ls > foo.txt
```

I/O Redirection

- **Answer: By calling the `dup2(oldfd, newfd)` function**
  - Makes **`newfd`** to be the copy of **`oldfd`**
  - Closing **`newfd`** first if necessary

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

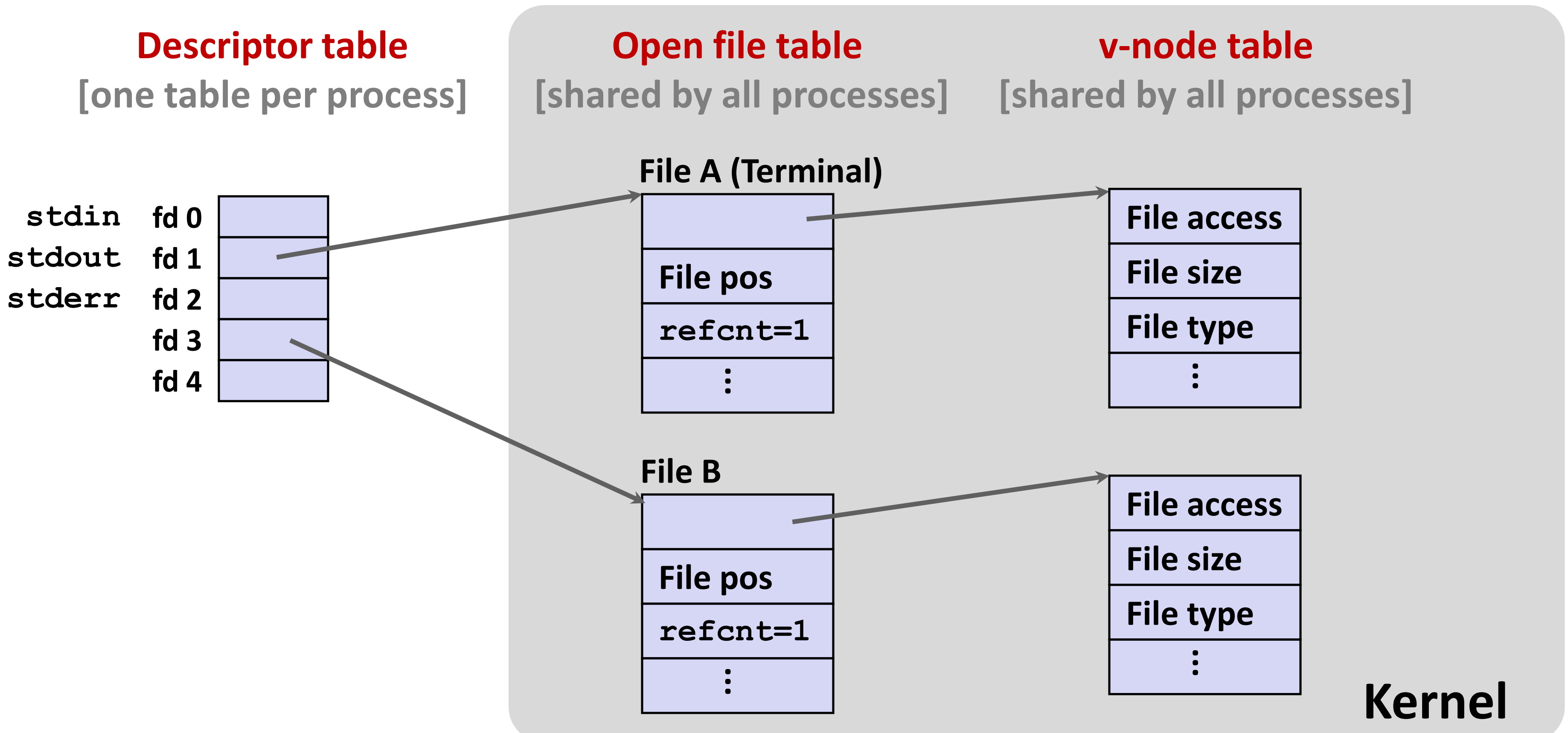
fd 0	
fd 1	`terminal`
fd 2	
fd 3	
fd 4	`foo.txt`

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

fd 0	
fd 1	`foo.txt`
fd 2	
fd 3	
fd 4	`foo.txt`

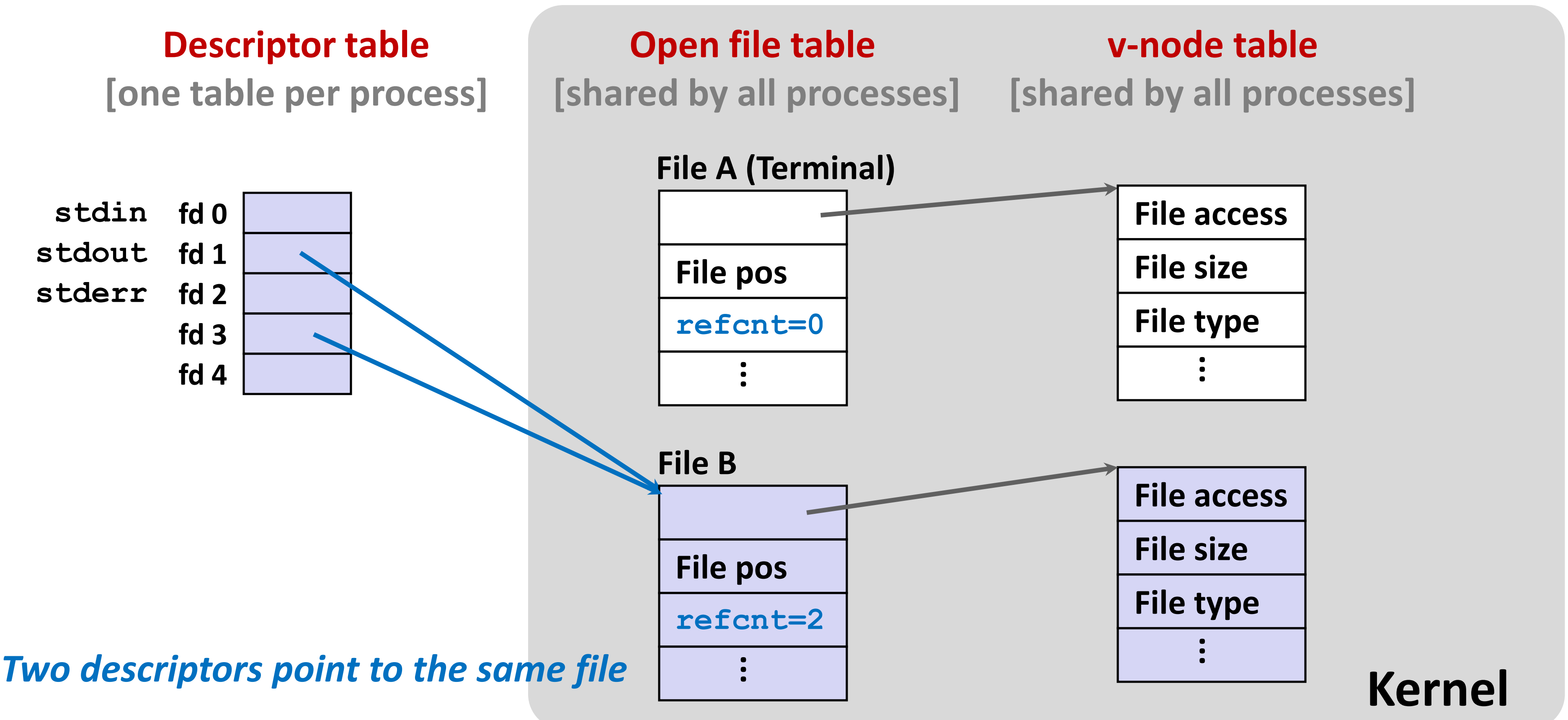
I/O Redirection Example

- **Step #1: open a file (fd=3) to redirect the stdout (fd=1)**
  - Happens in child executing shell code, before **`exec`**



I/O Redirection Example (cont.)

- **Step #2: call `dup2(3,1)`**
  - Cause fd=1 (**`stdout`**) points to the file pointed by fd=3



I/O Redirection Example (cont.)

```
$ strace -f bash -c "ls > foo.txt" 2>&1|grep "foo.txt" -n5
```

```
245:[pid   526] openat(AT_FDCWD, "foo.txt", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 3
246-[pid   526] dup2(3, 1)                  = 1
247-[pid   526] close(3)                    = 0
248-[pid   526] execve("/usr/bin/ls", ["ls"], 0x5613f6faeb40 /* 28 vars */) = 0
```

Warm-Up: I/O and Redirection Example

```
#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;
}                                                    ffiles1.c
```

- **Q. What would this program print for file containing "`abcde`"?**

Master Class: Process Control and I/O

```
#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;
}                                               ffiles2.c
```

- **Q. What would this program print for file containing "`abcde`"?**

Today

- Unix I/O
- Metadata, sharing, and redirection
- **Standard I/O**
- Closing remarks

Standard I/O Functions

- **The C standard library (`libc.so`) contains a collection of higher-level *standard I/O* functions**
  - Documented in Appendix B of K&R
  - https://man7.org/linux/man-pages/dir_section_3.html

- **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

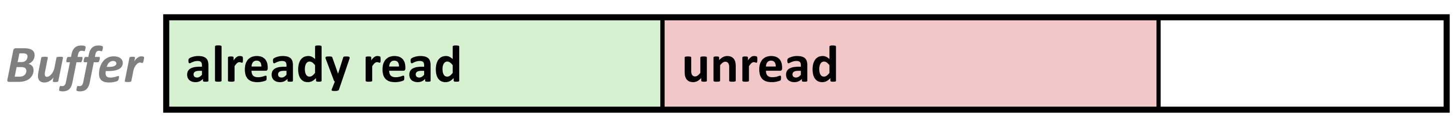
- **C programs begin life with three open streams (defined in `stdio.h`)**
  - `stdin` (standard input)
  - `stdout` (standard output)
  - `stderr` (standard error)

```
#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");
}
```

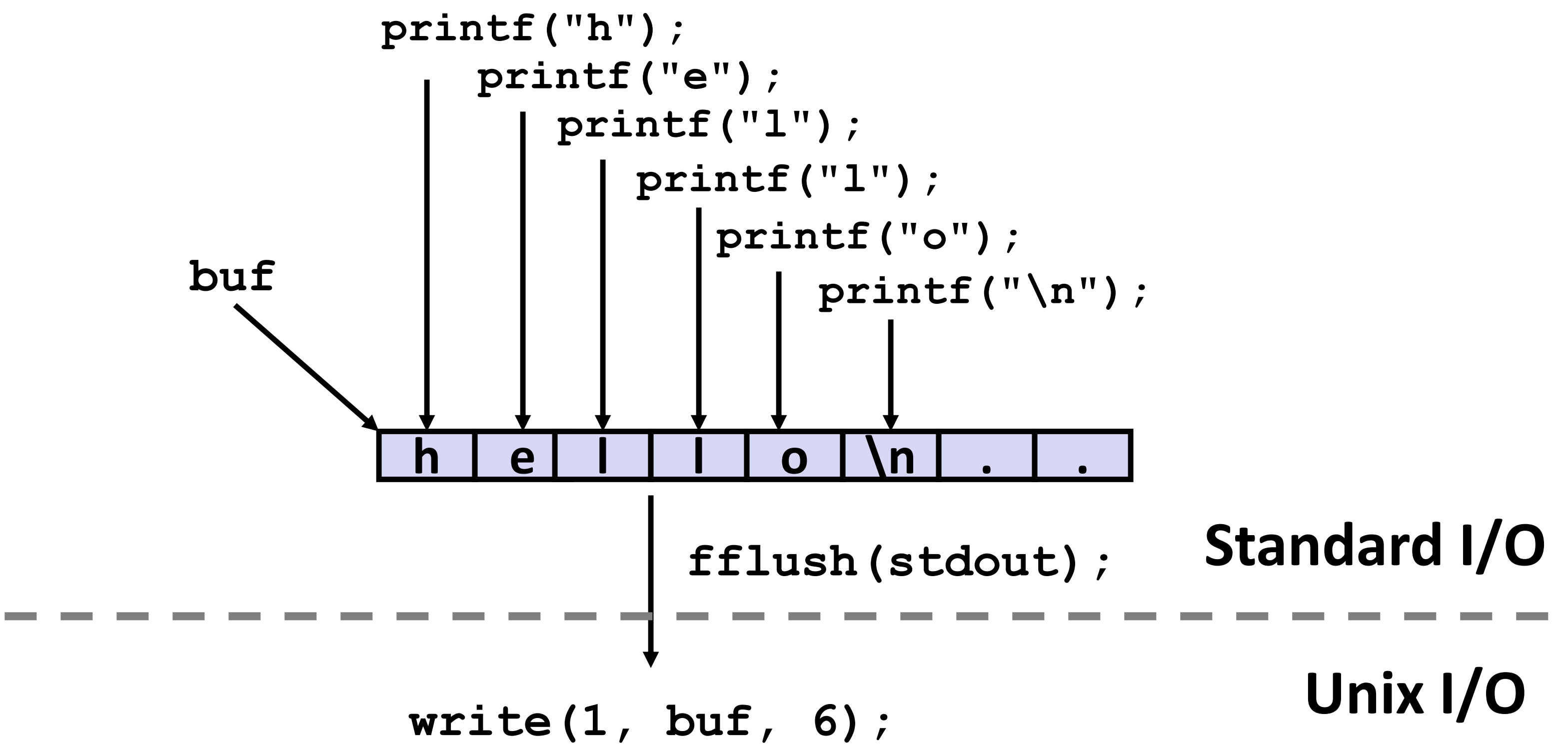
Buffered I/O: Motivation

- **Applications often read/write one character at a time**
  - **`getc, putc, ungetc`**
  - **`gets, fgets`**
    - Read line of text one character at a time, stopping at newline
- **Implementing as Unix I/O calls expensive**
  - **`read`** and **`write`** require kernel system calls
    - System call takes more than 10,000 clock cycles
- **Solution: 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



Buffering in Standard I/O

- Standard I/O functions use buffered I/O



- Buffer flushed to output fd on i) "\n", ii) call to `fflush`, iii) `exit`, or iv) return from `main`

Standard I/O Buffering in Action

- You can see this buffering in action for yourself, using the always fascinating Linux `strace` program:

```
#include <stdio.h>

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

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

Standard I/O Example

- **Dumping a file to `stdout`, line-by-line with stdio**

```
#define MLINE 1024

int main(int argc, char *argv[])
{
     char buf[MLINE];
     FILE *infile = stdin;
     if (argc == 2) {
          infile = fopen(argv[1], "r");
          if (!infile) exit(1);
     }
     while(fgets(buf, MLINE, infile) != NULL)
          fprintf(stdout, buf);
     exit(0);
}
```

showfile3_stdio.c

- **Demo:**

```
$ strace ./showfile3_stdio names.txt
```

- **Q. How many times of read() and write() would be invoked?**

Today

- Unix I/O
- Metadata, sharing, and redirection
- Standard I/O
- **Closing remarks**

Standard I/O Example

- **Dumping a file to stdout, loading entire file with mmap**

```
#include "csapp.h"


int main(int argc, char **argv)
{
    struct stat stat;
    if (argc != 2) exit(1);
    int infd = open(argv[1], O_RDONLY, 0);
    fstat(infd, &stat);
    size_t size = stat.st_size;
    char *bufp = mmap(NULL, size, PROT_READ,
                      MAP_PRIVATE, infd, 0);
    write(1, bufp, size);
    exit(0);
}
```

showfile5_mmap.c

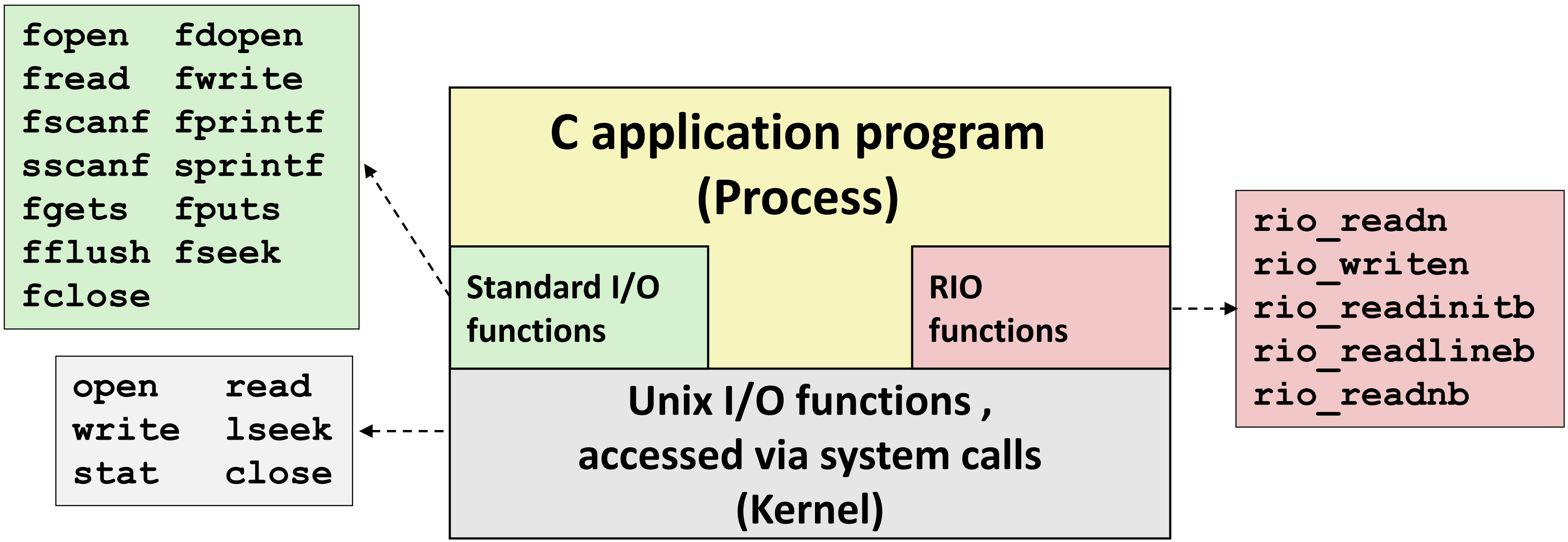
- **Demo:**

```
$ strace ./showfile5_mmap names.txt
```

- **What's the good of this dumping approach?**

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 functions are async-signal-safe and can be used safely in signal handlers

- **Cons**
  - Dealing with **short counts** is tricky and error prone (if you do it yourself)
  - Efficient reading of text lines requires some form of **buffering**, also tricky and error prone (if you do it yourself)
  - 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:**
  - Standard I/O functions are not async-signal-safe, and not appropriate for signal handlers
  - 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 (CS:APP3e, Sec 10.11)
    - Network streams (TCP/UDP) have unique characteristics different from file streams (such as **non-blocking I/O**)

Choosing I/O Functions

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

- **When to use standard I/O**
  - When working with disk or terminal files
- **When to use raw Unix I/O**
  - *Inside signal handlers, because Unix I/O is async-signal-safe*
  - In rare cases when you need absolute highest performance
- **When to use RIO**
  - *When you are reading and writing network sockets*
  - Avoid using standard I/O on sockets

For Further Information

- **The Unix bible:**
  - W. Richard Stevens & Stephen A. Rago, ***Advanced Programming in the Unix Environment***, 3rd Edition, Addison Wesley, 2013
    - Updated from Stevens's 1993 classic text

- **The Linux bible:**
  - Michael Kerrisk, The Linux Programming Interface, No Starch Press, 2010
    - Encyclopedic and authoritative