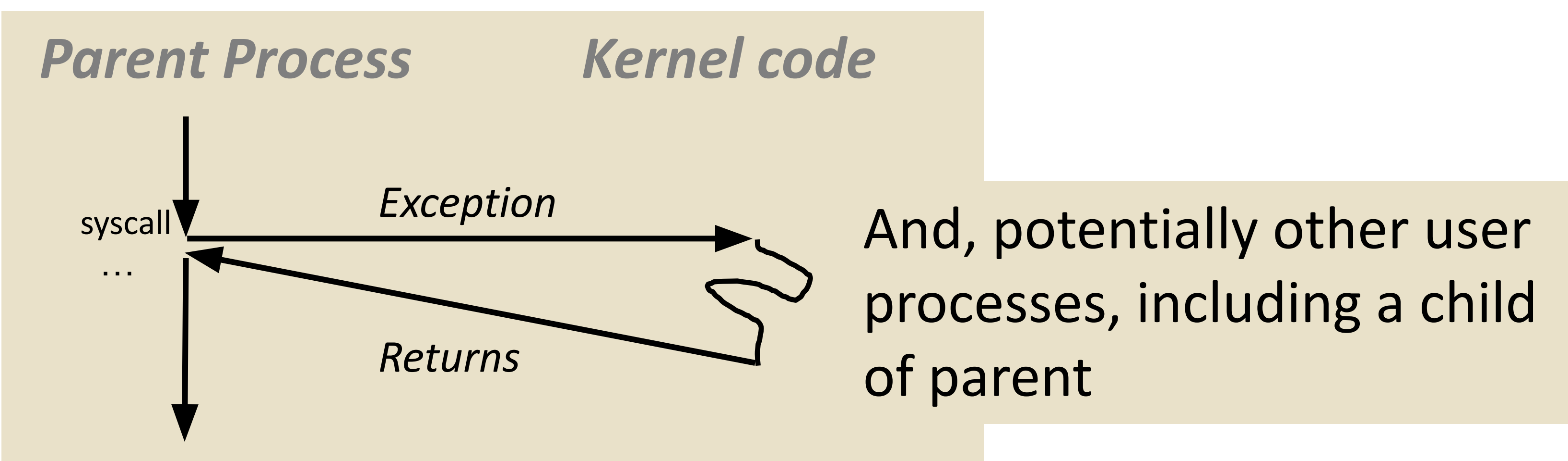
Process Control

`wait`: Synchronizing with Children

- **Parent reaps a child by calling the `wait` function**

- **`int wait(int *child_status)`**
  - Suspends current process until one of its children terminates
  - Implemented as syscall



`wait`: Synchronizing with Children

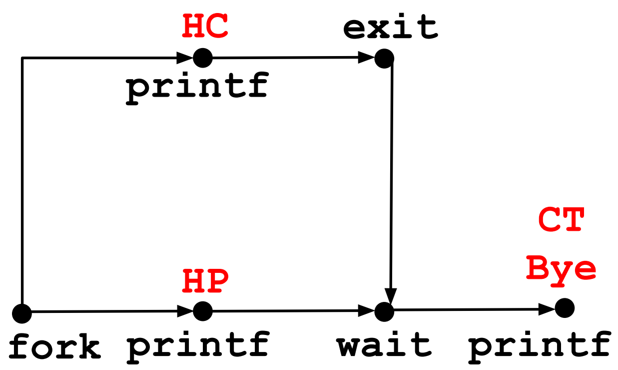
- **Parent reaps a child by calling the `wait` function**

- **`int wait(int *child_status)`**
  - Suspends current process until one of its children terminates
  - Return value is the **`pid`** of the child process that terminated
  - If **`child_status != NULL`**, then the integer it points to will be set to a value that indicates reason the child terminated and the exit status:
    - Checked using macros defined in `wait.h`
      - `WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG, WIFSTOPPED, WSTOPSIG, WIFCONTINUED`
      - See textbook for details

wait: Synchronizing with Children

```
void fork9() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
	exit(0);
    } else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
}
```

forks.c



Feasible output(s):

HC	HP
HP	HC
CT	CT
Bye	Bye

Infeasible output:

HP
CT
Bye
HC

Another wait Example

- If multiple children completed, will take in arbitrary order
- Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```
void fork10() {
    pid_t pid[N];
    int i, child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            exit(100+i); /* Child */
        }
    for (i = 0; i < N; i++) { /* Parent */
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```

forks.c

waitpid: Waiting for a Specific Process

- `pid_t waitpid(pid_t pid, int *status, int options)`
  - Suspends current process until specific process terminates
  - Various options (see textbook)

```
void fork11() {
    pid_t pid[N];
    int i;
    int child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = N-1; i >= 0; i--) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```

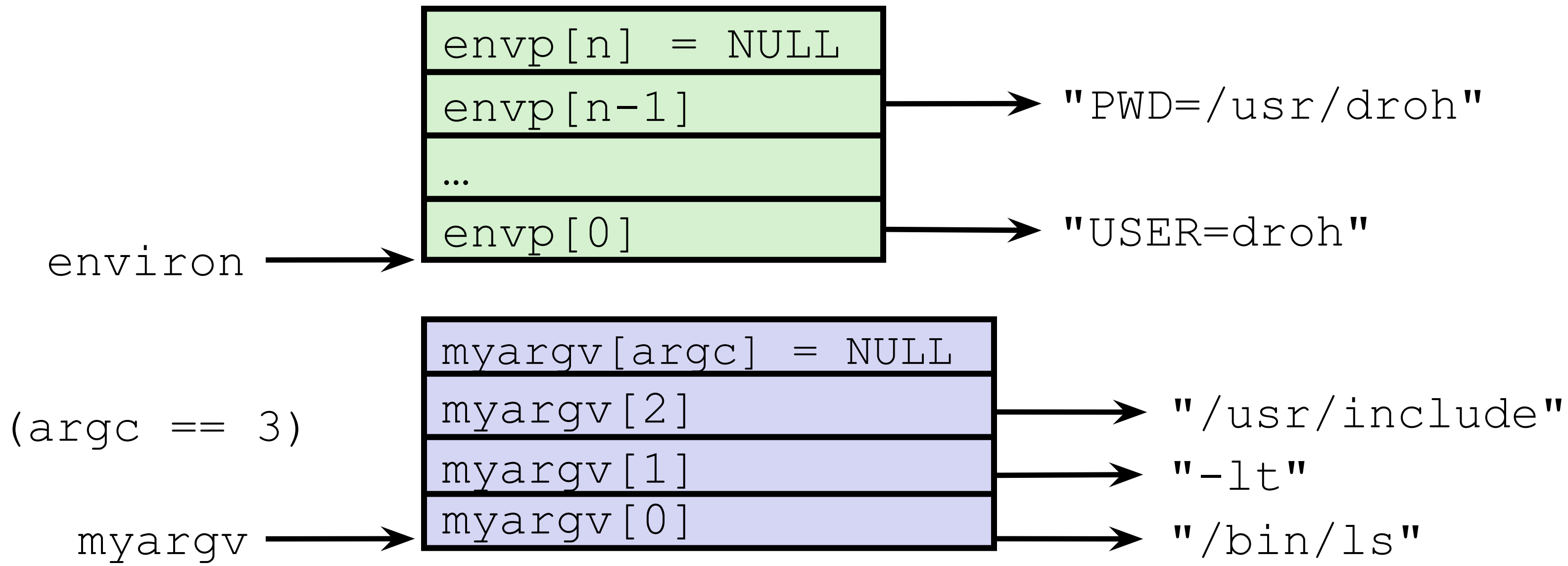
forks.c

`execve`: Loading and Running Programs

- `int execve(char *filename, char *argv[], char *envp[])`
- **Loads and runs in the current process:**
  - Executable file **`filename`**
    - Can be object file or script file beginning with `#!interpreter` (e.g., `#!/bin/bash`)
  - …with argument list **`argv`**
    - By convention **`argv[0]==filename`**
  - …and environment variable list **`envp`**
    - "name=value" strings (e.g., `USER=droh`)
    - `getenv, putenv, printenv`
- **Overwrites code, data, and stack**
  - Retains PID, open files and signal context
- **Called once and never returns**
  - …except if there is an error

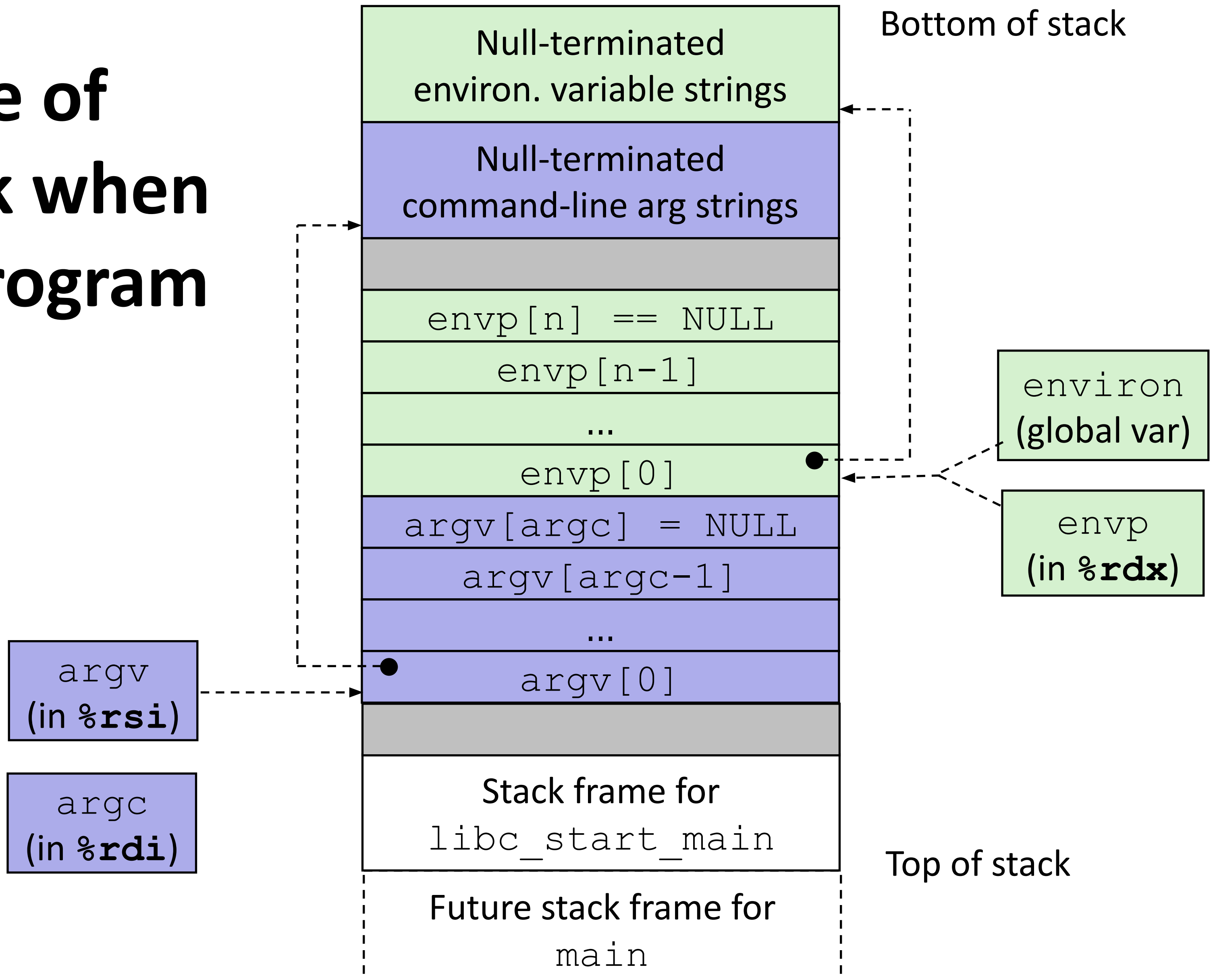
execve Example

- **Execute** `"/bin/ls -lt /usr/include"` **in child process using current environment:**

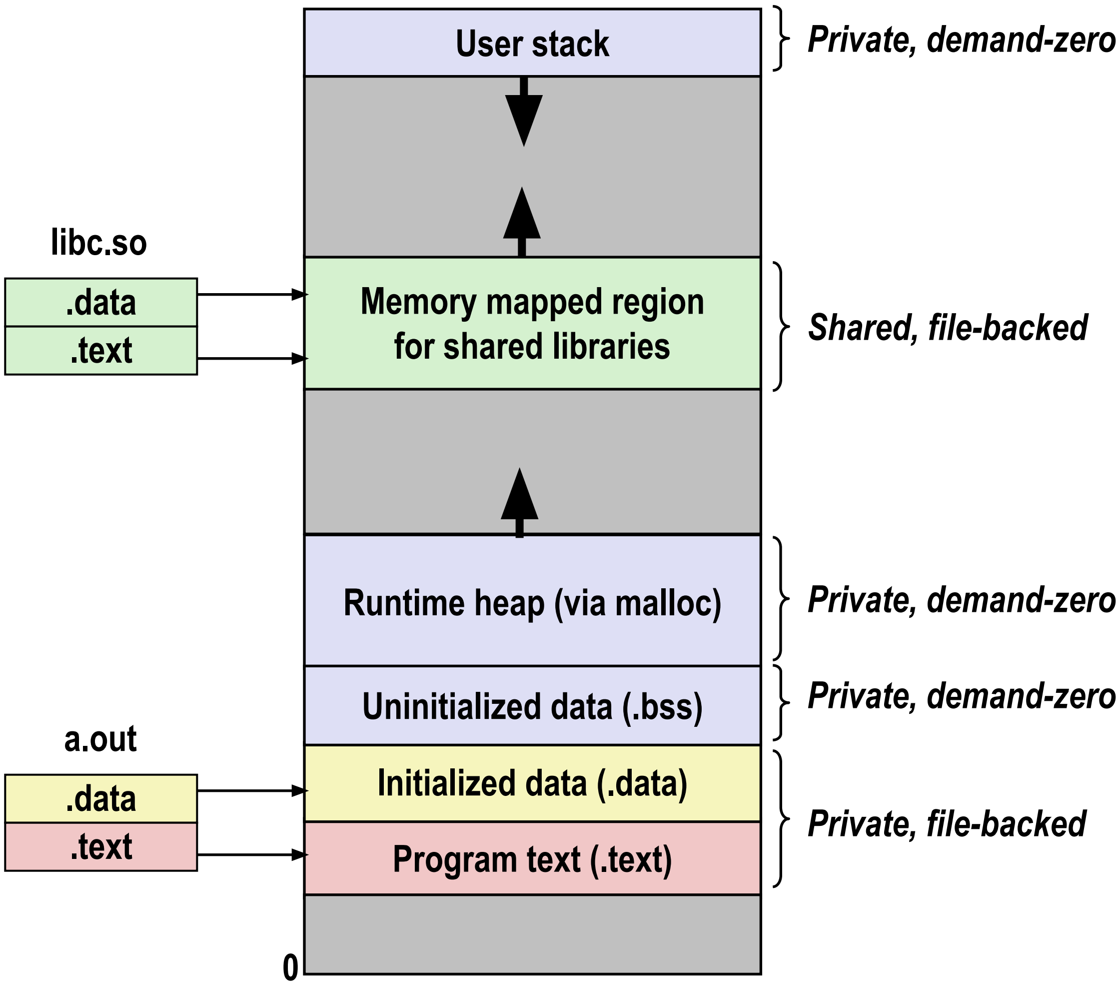



```
if ((pid = Fork()) == 0) {   /* Child runs program */
    if (execve(myargv[0], myargv, environ) < 0) {
        printf("%s: Command not found.\n", myargv[0]);
        exit(1);
    }
}
```

Structure of the stack when a new program starts



The `execve` Function Revisited



- To load and run a new program `a.out` in the current process using `execve`:
- Free `vm_area_struct`'s and page tables for old areas
- Create `vm_area_struct`'s and page tables for new areas
  - Programs and initialized data backed by object files.
  - `.bss` and stack backed by anonymous files.
- Set PC to entry point in `.text`
  - Linux will fault in code and data pages as needed.

System-Level I/O

System level: below standard level

```c
#include <stdio.h>

int main(void) {
    FILE *fp = fopen("output.txt", "w");
    if (!fp) {
        perror("output.txt");
        return 1;
    }
    fputs("baby shark (do doo dooo)\n", fp);
    if (fclose(fp)) {
        perror("output.txt");
        return 1;
    }
    return 0;
}
```

```c
FILE *fopen(const char *fname,
            const char *mode) {
    int fd = open(fname,
                  __mode2flags(mode),
                  DEFFILEPERMS);
    if (fd == -1) {
        return NULL;
    }
    return fdopen(fd, mode);
}
```

```c
int fputs(const char *s, FILE *fp) {
    size_t n = strlen(s);
    while (n > 0) {
        ssize_t written =
            write(fp->fd, s, n);
        if (written < 0) return EOF;
        n -= written;
        s += written;
    }
    return 0;
}
```

```c
int fclose(FILE *fp) {
    int rv = close(fp->fd);
    __ffree(fp);
    return rv;
}
```

```
.globl close
close:
    mov $3, %eax
    syscall
    cmp $-4096, %rax
    jae __syscall_error
    ret
```

Unix I/O Overview

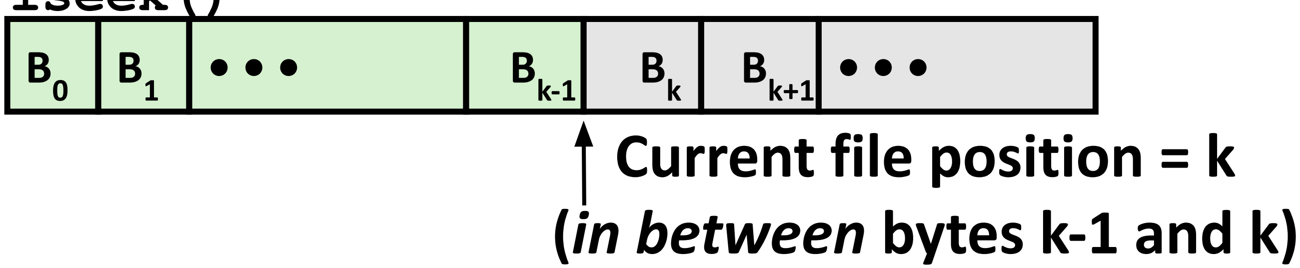
- A *file* is a sequence of bytes:
  - $B_0, B_1, \ldots., B_k, \ldots., B_{m-1}$

- All I/O devices are represented as files:
  - `/dev/sda2` (disk partition)
  - `/dev/tty2` (terminal)
  - `/dev/null` (discard all writes / read empty file)

- Kernel data structures are exposed as files
  - `cat /proc/$$/status`
  - `ls -l /proc/$$/fd/`
  - `ls –RC /sys/devices | less`

Unix I/O Overview

- **Kernel offers a set of basic operations for all files**
  - Opening and closing files
    - `open()` and `close()`
  - Reading and writing a file
    - `read()` and `write()`
  - Look up information about a file (size, type, last modification time, …)
    - `stat()`, `lstat()`, `fstat()`
  - Changing the ***current file position*** (seek)
    - indicates next offset into file to read or write
    - `lseek()`



File Types

- **Each file has a *type* indicating its role in the system**
  - *Regular file:* Stores 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

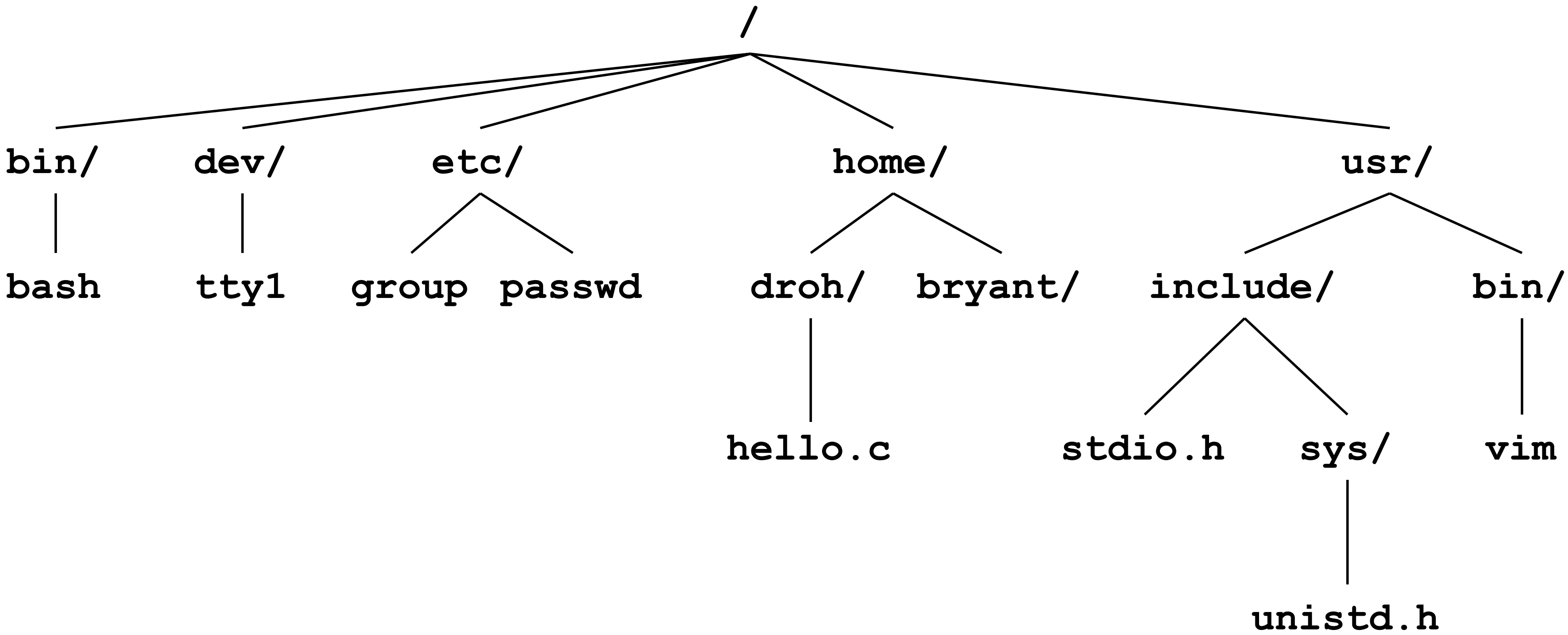
- **A regular file contains arbitrary data**
- **Applications often distinguish between *text* and *binary files***
  - Text files contain human-readable text
  - Binary files are everything else (object files, JPEG images, …)
  - Kernel doesn't care! It's all just bytes!
- **Text file is sequence of *text lines***
  - Text line is sequence of characters terminated (not separated!) by *end of line indicator*
  - Characters are defined by a *text encoding* (ASCII, UTF-8, EUC-JP, …)
- **End of line (EOL) indicators:**
  - All "Unix": Single byte **`0x0A`**
    - line feed (LF)
  - DOS, Windows: Two bytes **`0x0D 0x0A`**
    - Carriage return (CR) followed by line feed (LF)
    - Also used by many Internet protocols
  - C library translates to '\n'

Directories

- **Directory consists of an array of *entries* (also called *links*)**
  - Each entry maps a *filename* to a file
- **Each directory contains at least two entries**
  - `.` (dot) maps to the directory itself
  - `..` (dot dot) maps 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

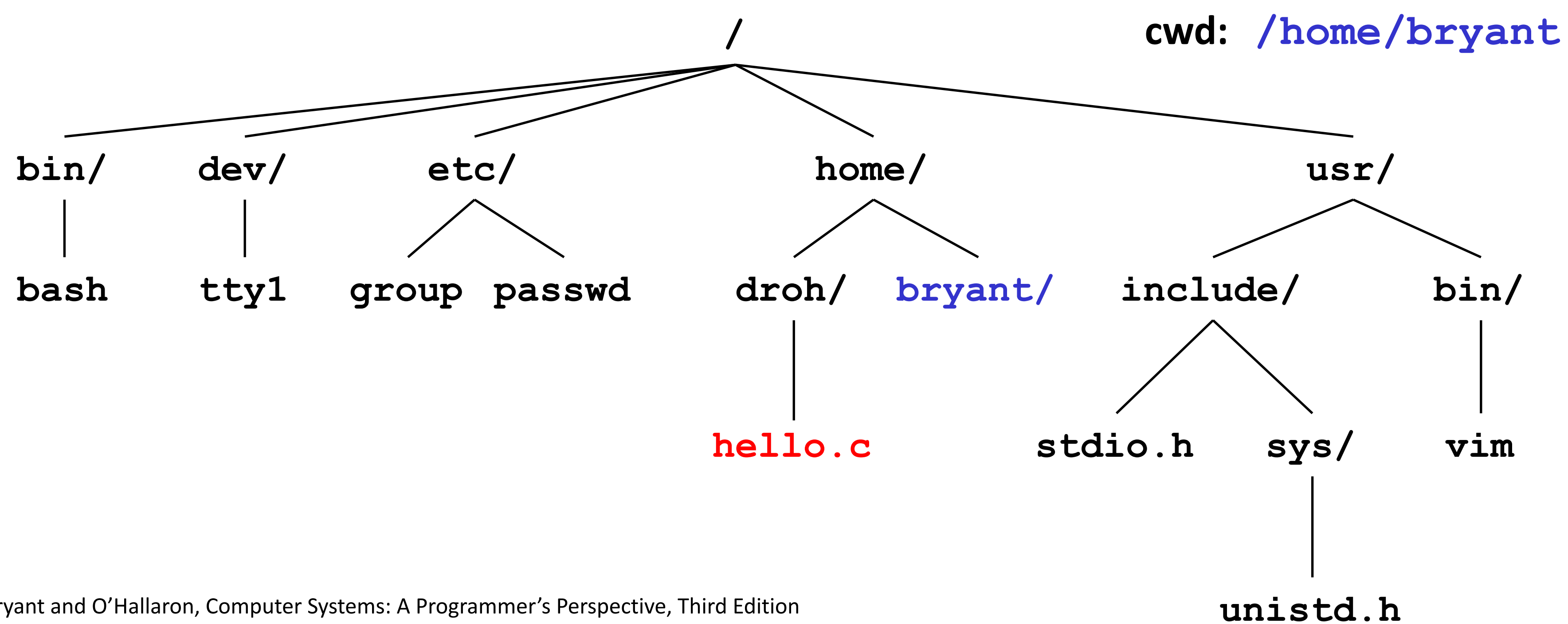
- All files are organized as a hierarchy anchored by root directory named / (slash)



- 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/droh/hello.c`**
  - *Relative pathname* denotes path from current working directory
    - **`../droh/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**
  - 0: standard input (stdin)
  - 1: standard output (stdout)
  - 2: standard error (stderr)
  - These could be files, pipes, your terminal, or even a network connection!

Lots of ways to call `open`

Open an existing file:

`open(path, flags)`

`flags` must include exactly one of:

`O_RDONLY`	Only want to read from file
`O_WRONLY`	Only want to write to file
`O_RDWR`	Want to do both

Flags may also include (use | to combine)

`O_APPEND`	All writes go to the very end
`O_TRUNC`	Delete existing contents if any
`O_CLOEXEC`	Close this file if execve() is called

Open or create a file:

`open(path, flags, mode)`

`flags` must include

`O_CREAT`	Create the file if it doesn't exist

and exactly one of:

`O_WRONLY`	Only want to write to file
`O_RDWR`	Want to write and read

and maybe also some of:

`O_EXCL`	Fail if file does exist
`O_APPEND`	All writes go to the very end
`O_TRUNC`	Delete existing contents if any
`O_CLOEXEC`	Close this file if execve() is called

(and many more… consult the open() manpage)

The third argument to open

- **Yes, open takes either two or three arguments**
  - Look through `/usr/include/fcntl.h` and try to figure out how it's done
  - Third argument must be present when O_CREAT appears in second argument; ignored otherwise
- **Third argument gives *default access permissions* for newly created files**
  - Modified by *umask* setting (see `man umask`)
  - Use DEFFILEMODE (from `sys/stat.h`) unless you have a specific reason to want something else
  - More explanation:
    - https://linuxfoundation.org/blog/classic-sysadmin-understanding-linux-file-permissions/
    - https://linuxcommand.org/lc3_lts0090.php
    - https://devconnected.com/linux-file-permissions-complete-guide/

Closing Files

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

```
if (close(fd) < 0) {
   fprintf(stderr, "%s: write error: %s",
           filename, strerror(errno));
   exit(1);
}
```

- **Take care not to close any file more than once**
  - Same as not calling free() twice on the same pointer

- **Closing a file can fail!**
  - The OS is taking this opportunity to report a *delayed error* from a previous write operation
  - You might silently lose data if you don't check!

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 stdin to stdout, one byte at a time**

```
#include <unistd.h>

int main(void) {
    char c;
    while(read(STDIN_FILENO, &c, 1) != 0)
        write(STDOUT_FILENO, &c, 1);
    return 0;
}
```

Always check return codes from system calls!

Simple Unix I/O example

- **Copying stdin to stdout, one byte at a time**

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

int main(void) {
    char c;
    for (;;) {
        ssize_t nread = read(STDIN_FILENO, &c, 1);
        if (nread == 0) {
            return 0;
        } else if (nread < 0) {
            perror("stdin");
            return 1;
        }
        if (write(STDOUT_FILENO, &c, 1) < 1) {
            perror("stdout: write error");
            return 1;
        }
    }
}
```

Simple Unix I/O example

- **Copying stdin to stdout, one byte at a time**

```
#include "csapp.h"

int main(void) {
    char c;
    while (Read(STDIN_FILENO, &c, 1) != 0) {
        Write(STDOUT_FILENO, &c, 1);
    }
    return 0;
}
```

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, pipes, etc.

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

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

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

- **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

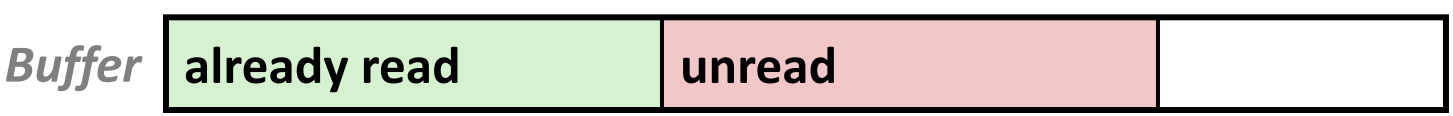
- **Standard I/O models open files as *streams***
  - Abstraction for a file descriptor and a buffer in memory

- **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");
}
```

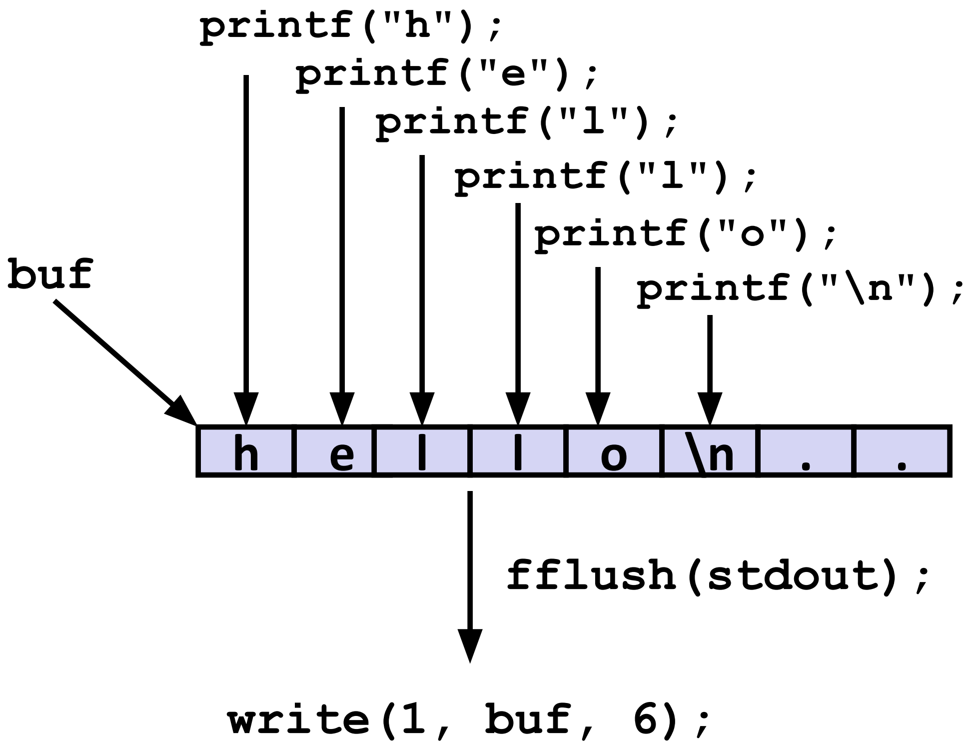
Buffered I/O: Motivation

- **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 Unix kernel calls
    - > 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 "\n", call to `fflush` or `exit`, or 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);
}
```

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

Pros and Cons of Unix I/O

- **Pros**
  - Unix I/O is the most general 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
  - 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
  - Efficient reading of text lines requires some form of buffering, also tricky and error prone

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 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)

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

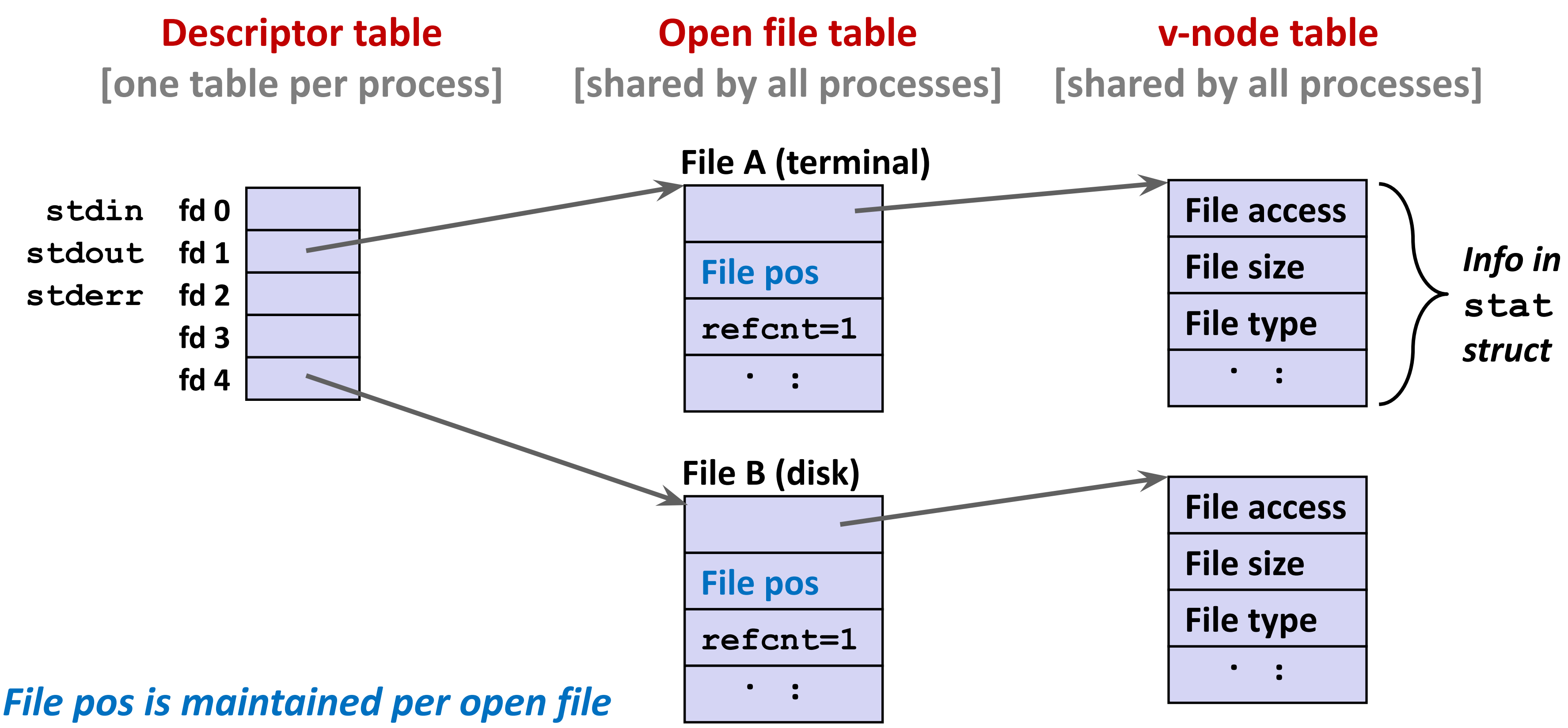
```
int main (int argc, char **argv)
{
    struct stat stat;
    char *type, *readok;

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

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

statcheck.c

```
linux> ./statcheck statcheck.c
type: regular, read: yes
linux> chmod 000 statcheck.c
linux> ./statcheck statcheck.c
type: regular, read: no
linux> ./statcheck ..
type: directory, read: yes
```

How the Unix Kernel Represents Open Files

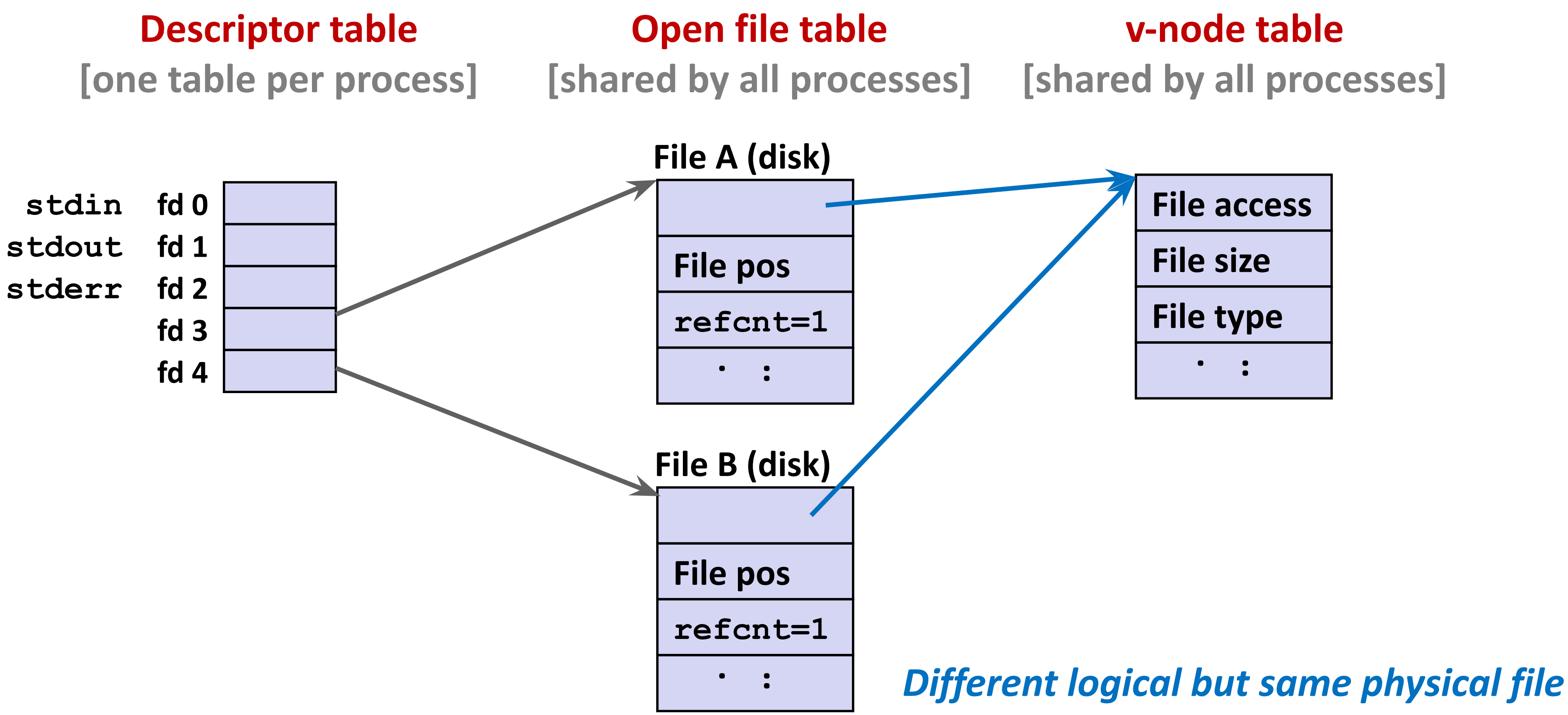
- Two descriptors referencing two distinct open files. Descriptor 1 (stdout) points to terminal, and descriptor 4 points to open disk file



File pos is maintained per open file

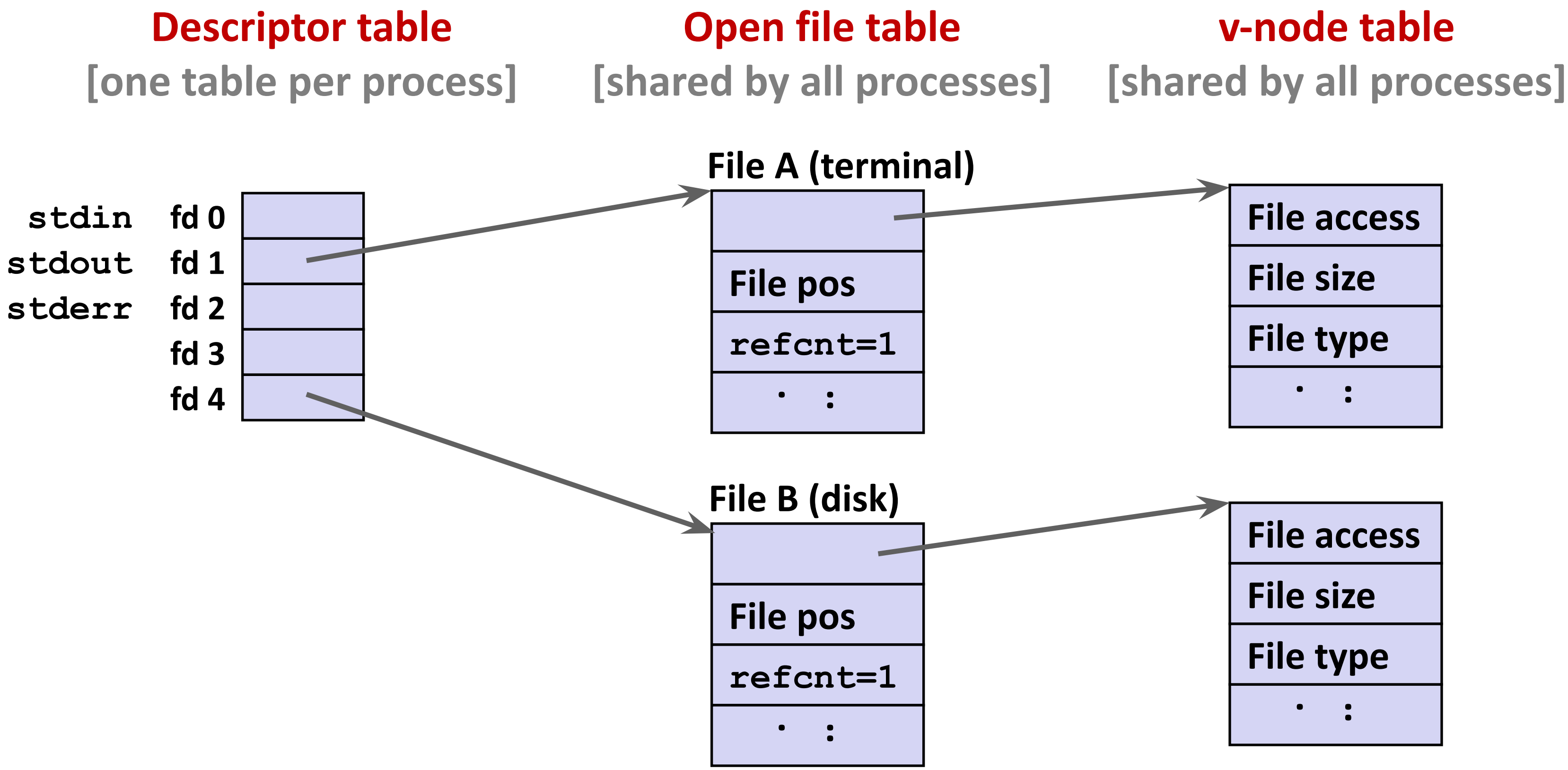
File Sharing

- **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



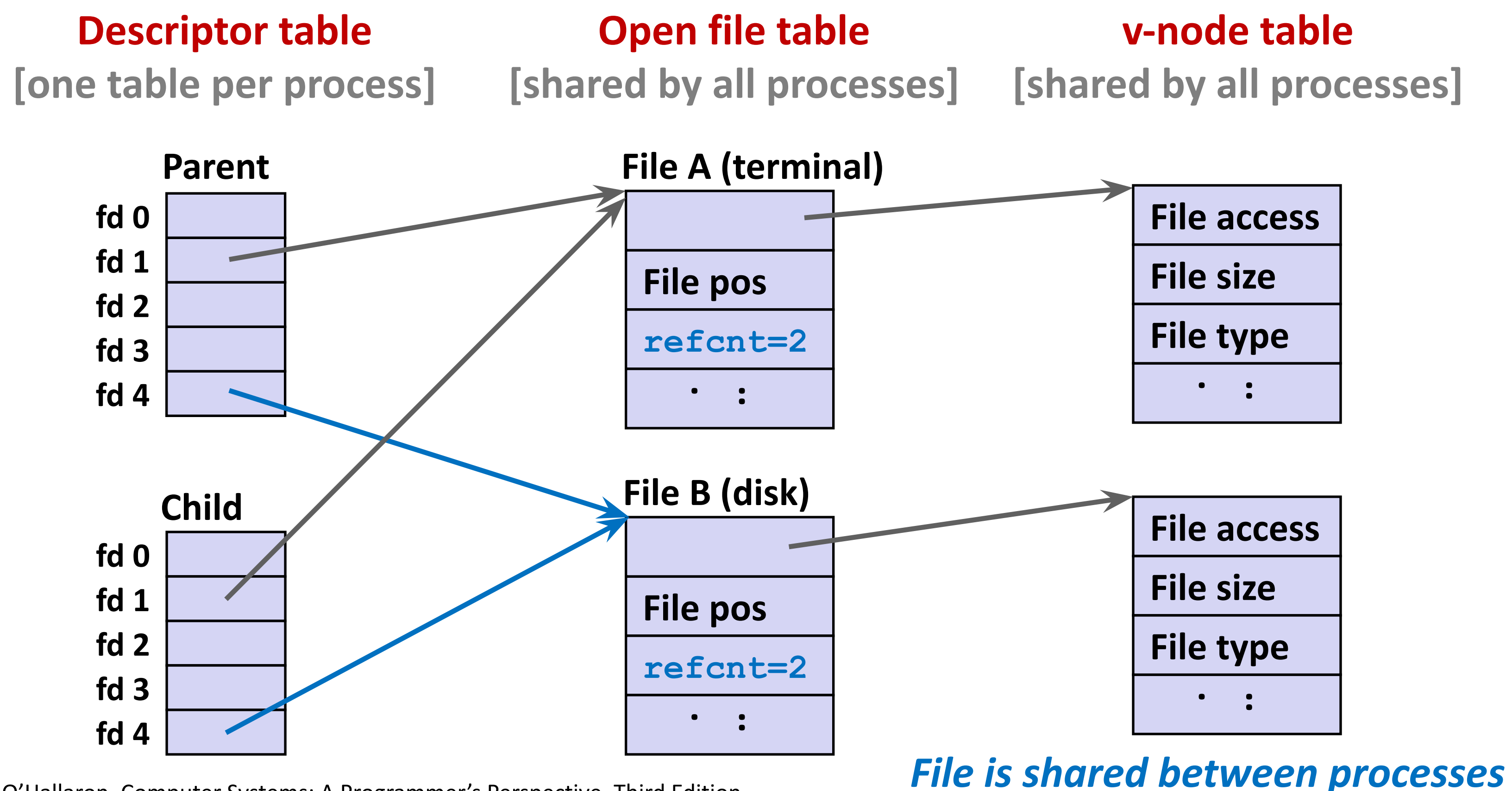
How Processes Share Files: `fork`

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

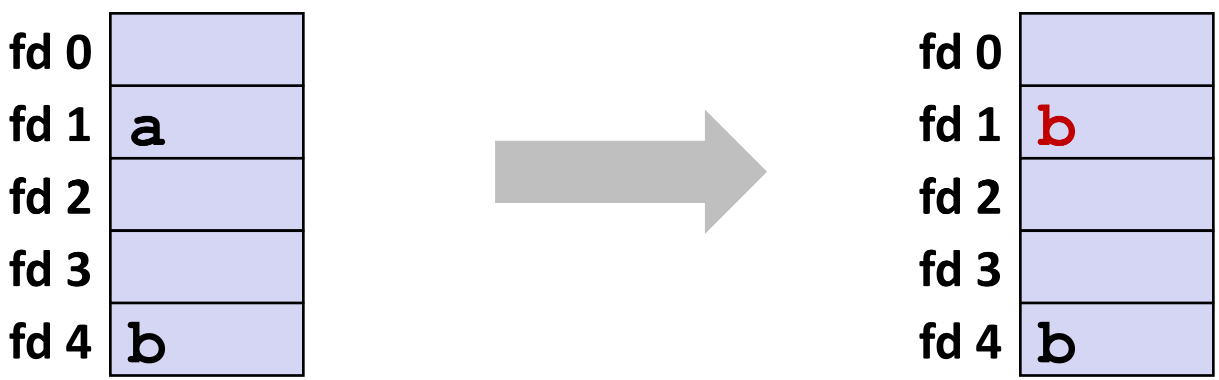
- **Question: How does a shell implement I/O redirection?**

  `linux> ls > foo.txt`

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

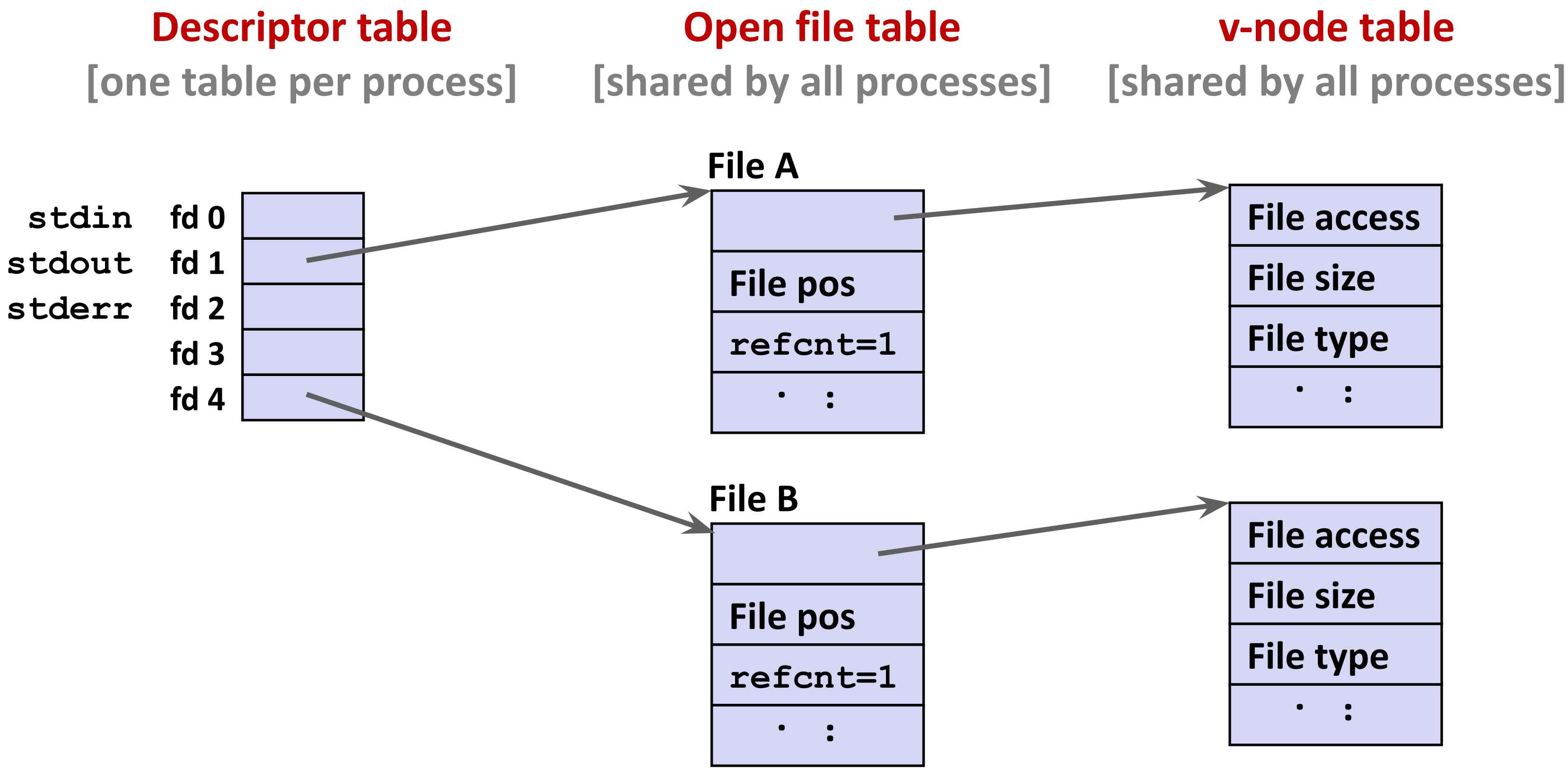
Descriptor table ***before*** **`dup2(4,1)`**

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



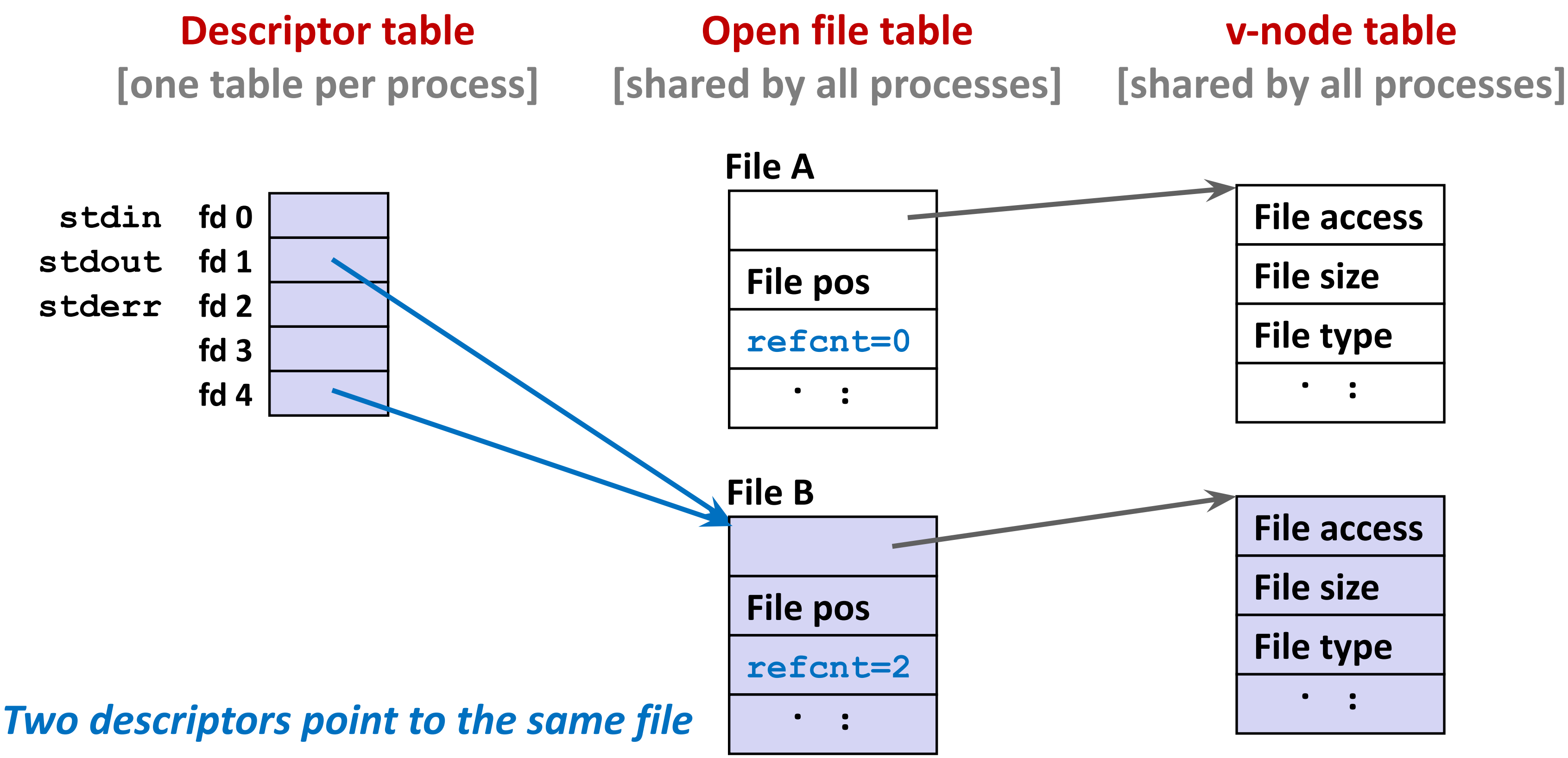
I/O Redirection Example

- **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



Problem: 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;
}
```

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