System-Level I/O

Read Chap 10.1-10.4, 10.6-10.12
Optional: Chap 10.5

Instructor: Jennifer Wong-Ma
jwongma@uci.edu

Input/Output (I/O)

- **Process of copying data between main memory & external sources**
  - Ex: Disk drives, Terminals, Networks
- **Languages provide high-level functions for I/O**
  - ANSI C has standard I/O (printf, scanf)
  - C++ has overloaded << and >> operators
- **Why do you need to know Unix I/O then?**
  - Better understanding of the system and how it operates
  - Sometimes, must use System I/O
    - Ex: file metadata, network programming

Unix I/O Overview

- A Linux *file* is a sequence of *m* bytes:
  - $B_0, B_1, \ldots, B_k, \ldots, B_{m-1}$
- Read and write to *file*; a mapping to I/O
- Cool fact: All I/O devices are represented as files:
  - `/dev/sda2` (`/usr` disk partition)
  - `/dev/tty2` (terminal)
- Even the kernel is represented as a file:
  - `/boot/vmlinuz-3.13.0-55-generic` (kernel image)
  - `/proc` (kernel data structures)

Unix I/O Overview – Open & Close

- Elegant mapping of files to devices allows kernel to export simple interface called *Unix I/O:*
  - `int open(char * filename, int flags, mode_t mode);`
    - Process asks kernel to access I/O device
    - Kernel returns descriptor – small nonnegative number
    - Kernel keeps track of all info associated with file, Process keeps track of descriptor
    - Returns: new file descriptor if OK, -1 on error
    - Flags – how to access the file:
      - O_RDONLY, O_WRONLY, O_RDWR
      - OR'd with O_CREAT, O_TRUNC, O_APPEND
    - Mode – access permission bits of new files
      - sys/stat.h

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 Linux shell begins life with three open files associated with a terminal: <unistd.h>**
  - 0: standard input (stdin – STDIN_FILENO)
  - 1: standard output (stdout – STDOUT_FILENO)
  - 2: standard error (stderr – STDERR_FILENO)

Unix I/O Overview – Open & Close

- `int close(int fd)`
  - Process informs kernel to close file
  - Kernel frees data stuctures associated with file and returns descriptor to pool
  - If process “dies” (any reason), kernel closes all open files and frees memory
  - Returns: 0 if OK, -1 on error

Closing Files

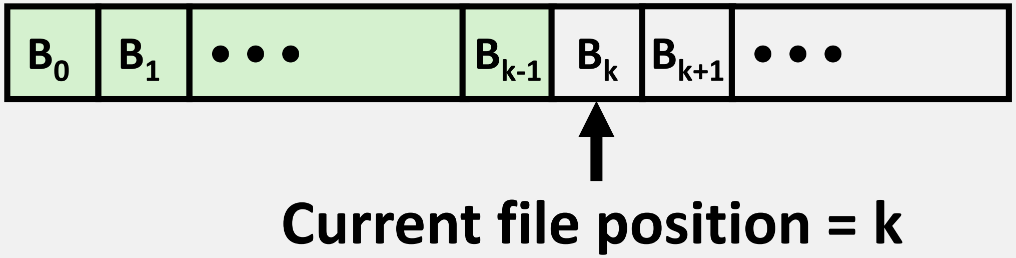
- Closing a file informs the kernel that you are finished accessing that file

```
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()`

Unix I/O Overview - Seek

- **Changing the *current file position* (seek)**
  - Starts at 0 for each open file
  - Indicates next offset into file to read or write
  - **`lseek()` - reposition read/write file offset**



Unix I/O Overview

- Reading and writing a file
  - `ssize_t read(int fd, void *buf, size_t n);`
    - Copies *n* > 0 bytes from file to memory from file position, *k*
    - Moves *k* by *n* bytes
    - If *k* > file size, triggers end-of-file (EOF) condition – no explicit "EOF character" at the end of the file
    - Returns: number of bytes read if OK, 0 on EOF, -1 on error
  - `ssize_t write(int fd, const void *buf, size_t n);`
    - Copies *n* > 0 bytes from memory to a file starting at file position, *k*
    - Moves *k* by *n* bytes
    - Returns: number of bytes written if OK, -1 on error

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 "csapp.h"

int main(void)
{
    char c;

    while(Read(STDIN_FILENO, &c, 1) != 0)
        Write(STDOUT_FILENO, &c, 1);
    exit(0);
}
```

On Short Counts

- **When read and write transfer fewer bytes than requested**
- **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
- **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.**

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

- **Named pipes (FIFOs)**
  - Named pipes exist as a device special file in the file system.
  - Processes of different ancestry can share data through a named pipe.
  - When all I/O is done by sharing processes, the named pipe remains in the file system for later use.

- **Symbolic links**
  - A special type of file that serves as a reference to another file or directory.

Regular Files

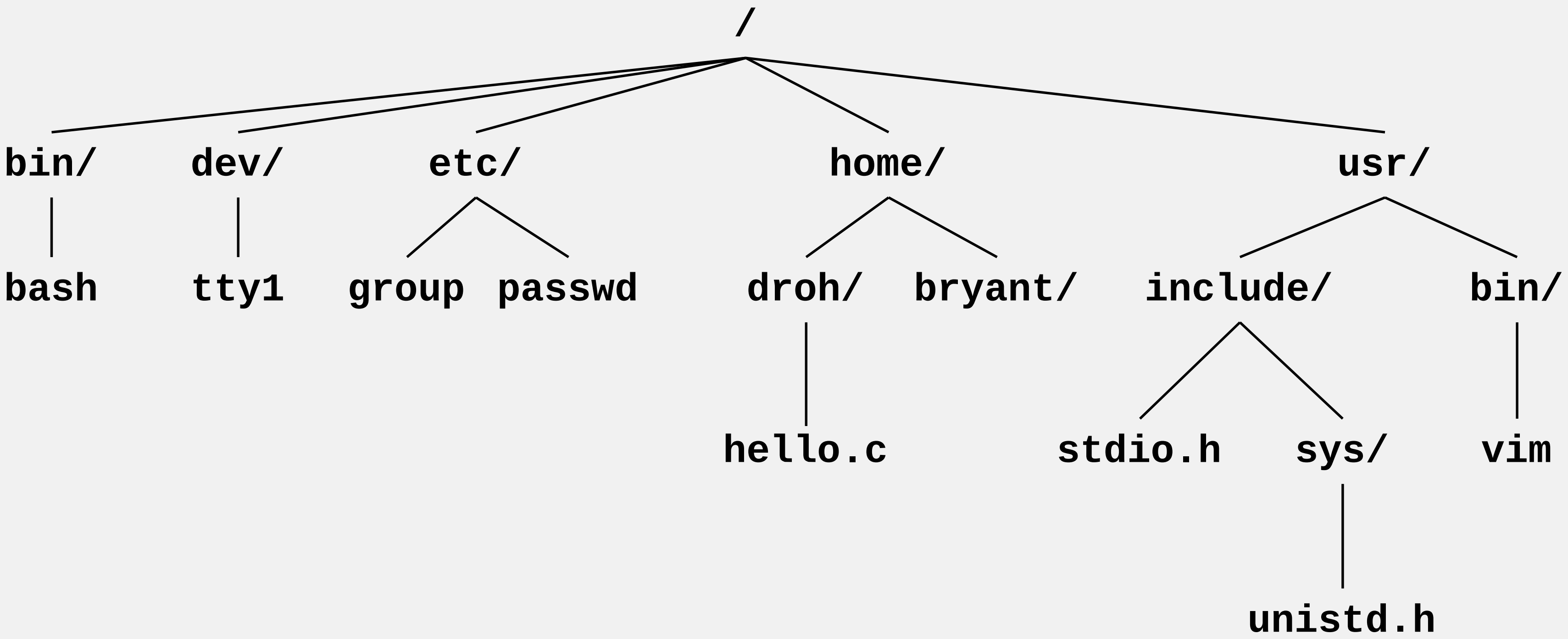
- **A regular file contains arbitrary data**
- **Applications often distinguish between *text files* and *binary files***
  - Text files are regular files with only ASCII or Unicode characters
  - Binary files are everything else
    - e.g., object files, JPEG images
  - Kernel doesn't know the difference!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

Directories

- **Directory consists of an array of *links***
  - Each link maps a *filename* 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

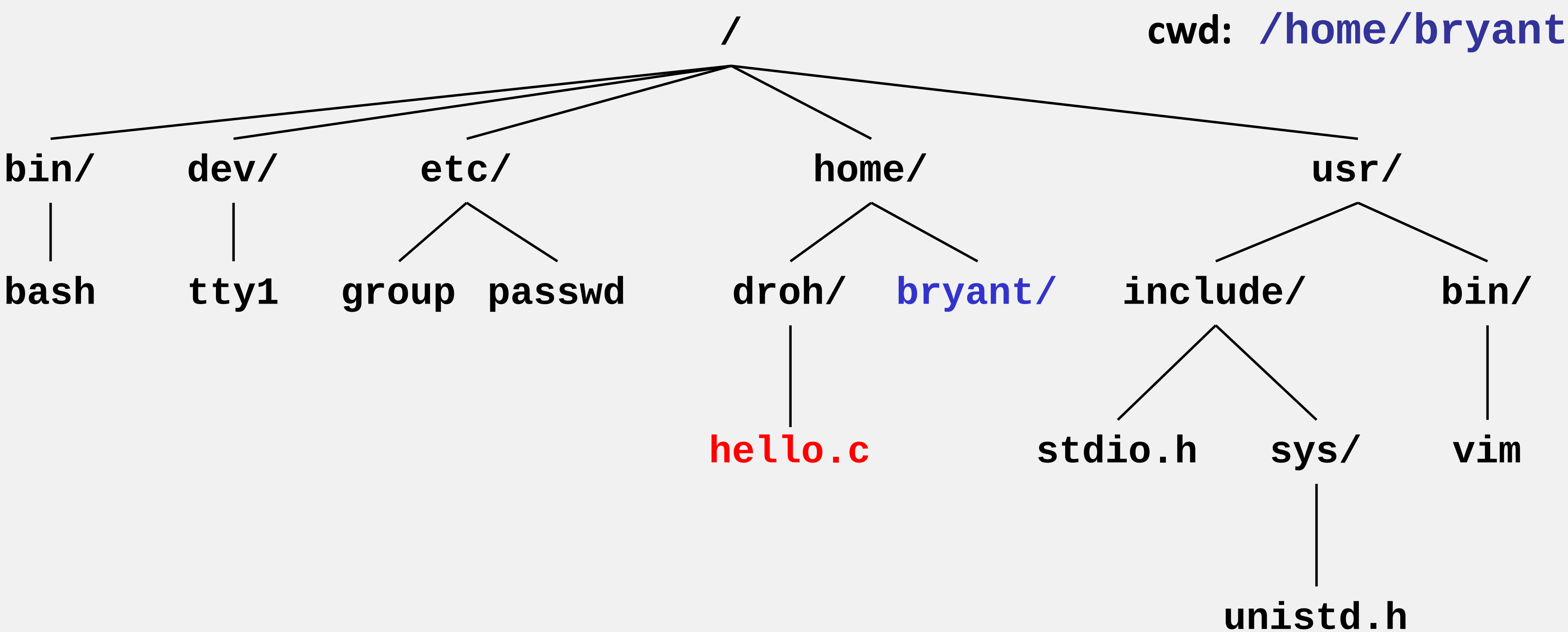
- **All files are organized as a hierarchy anchored by root directory named / (slash)**



- **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/droh/hello.c`
  - ***Relative pathname* denotes path from current working directory**
    - `../home/droh/hello.c`



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 */
};
```

File Metadata

- `int stat(const char *filename, struct stat *buf);`
  - Uses filename
- `int fstat(int fd, struct stat *buf);`
  - Uses file descriptor

  - Fills buf structure
  - Returns: 0 f OK, -1 on error

Example of Accessing File Metadata

```
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
```

```
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);
}
```

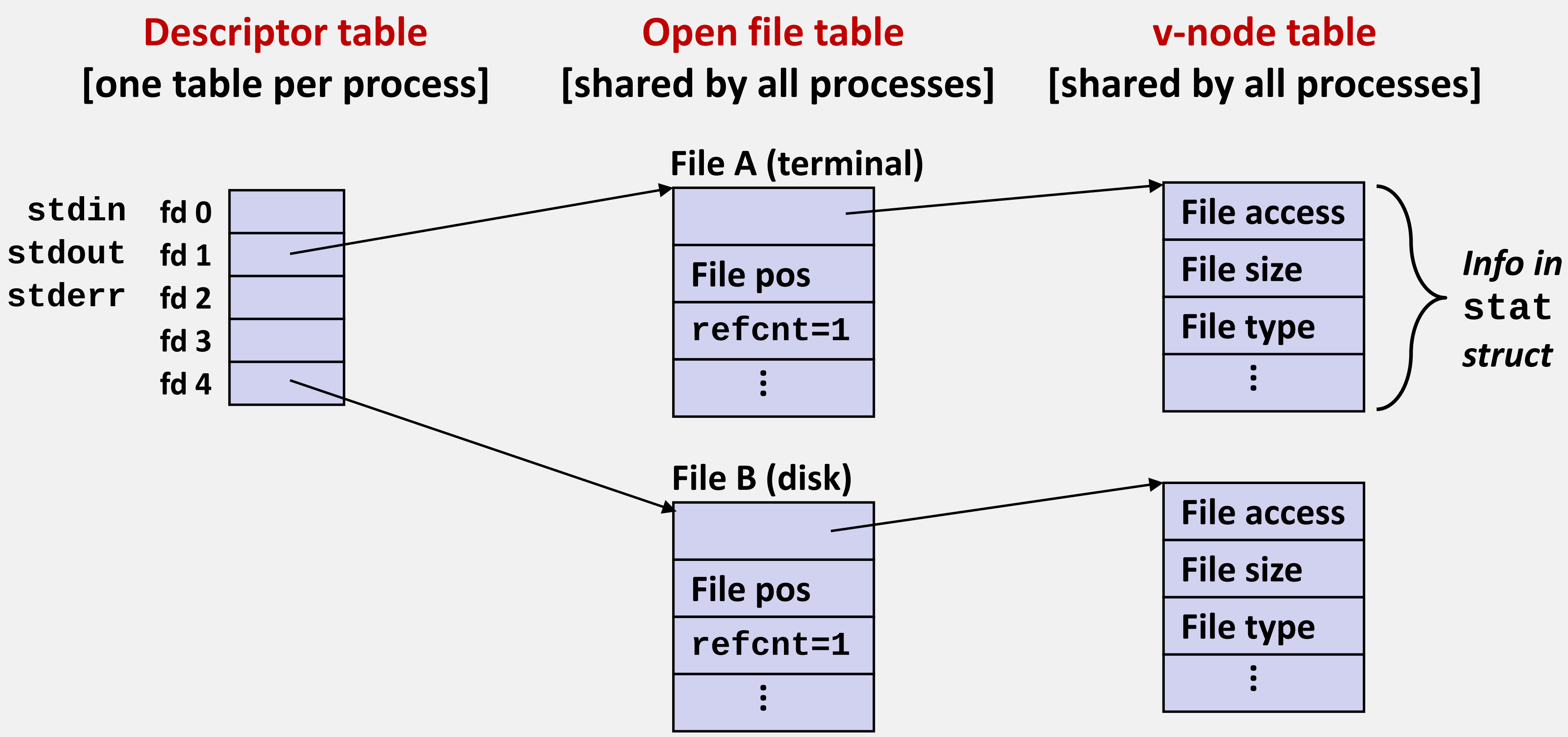
statcheck.c

File Sharing

- **Information about the files are kept via 3 levels:**
  - <u>Descriptor Table</u>
    - Each process has own table
    - Entries are index of file descriptors
    - Each open descriptor entry points to an entry in the file table
  - <u>File Table</u>
    - Shared by all processes
    - Entries contain information about file (current position, # of references, pointer to an entry in v-node table, etc)
    - Kernel only removes entry if reference count = 0
  - <u>V-node Table</u>
    - Shared by all processes
    - Each entry contains info from stat structure

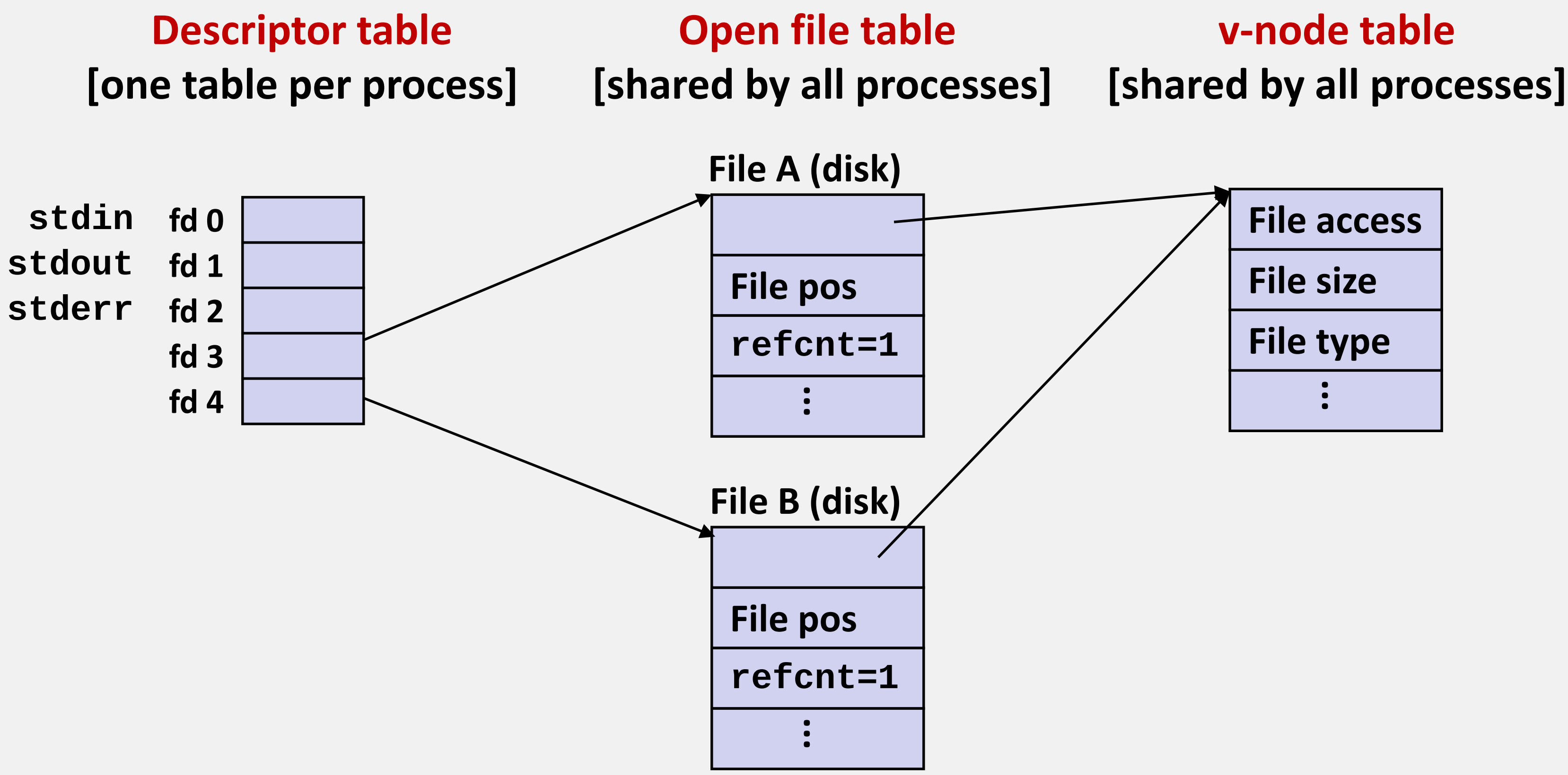
How the Unix Kernel Represents Open Files

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



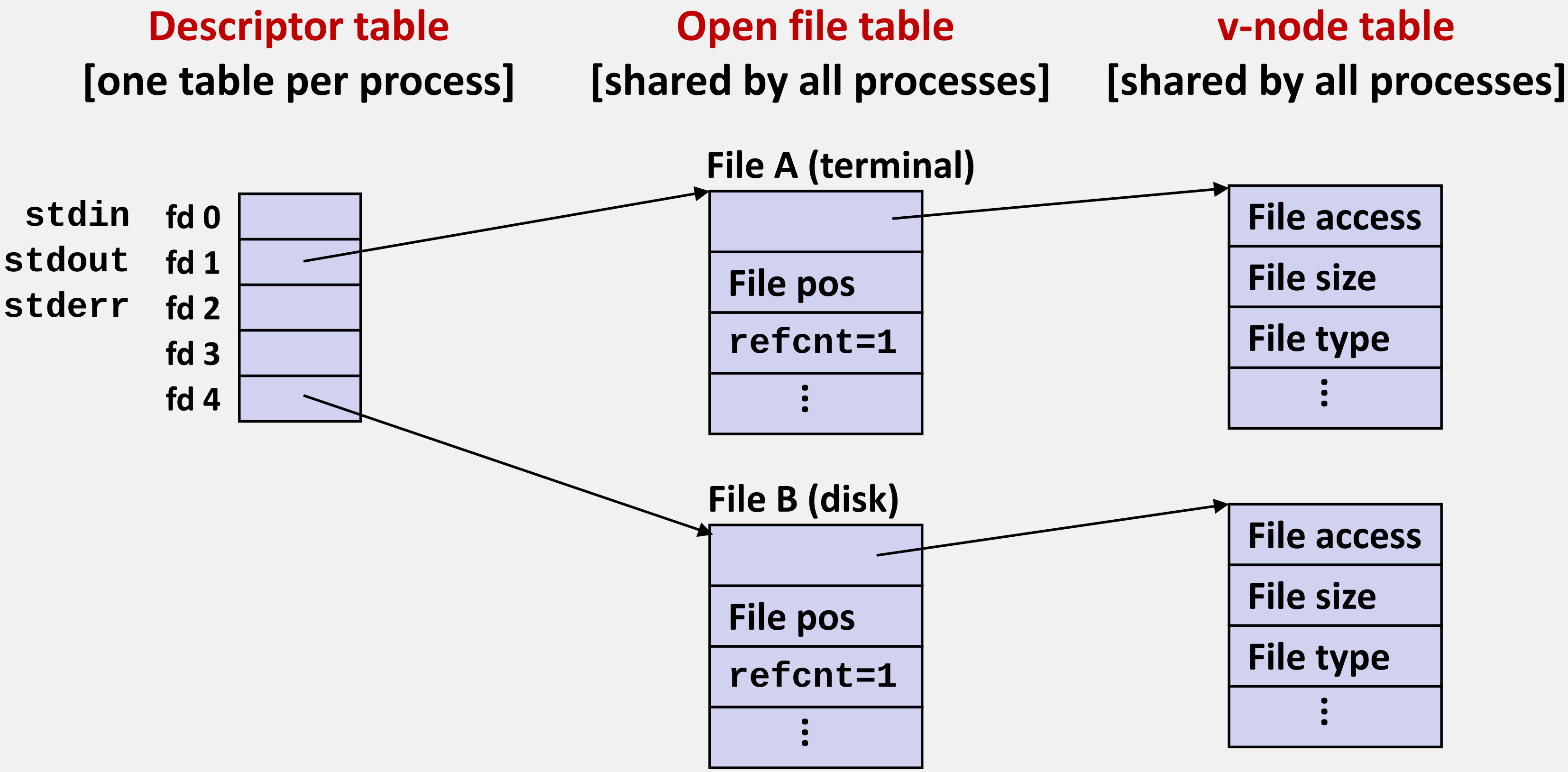
File Sharing

- **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



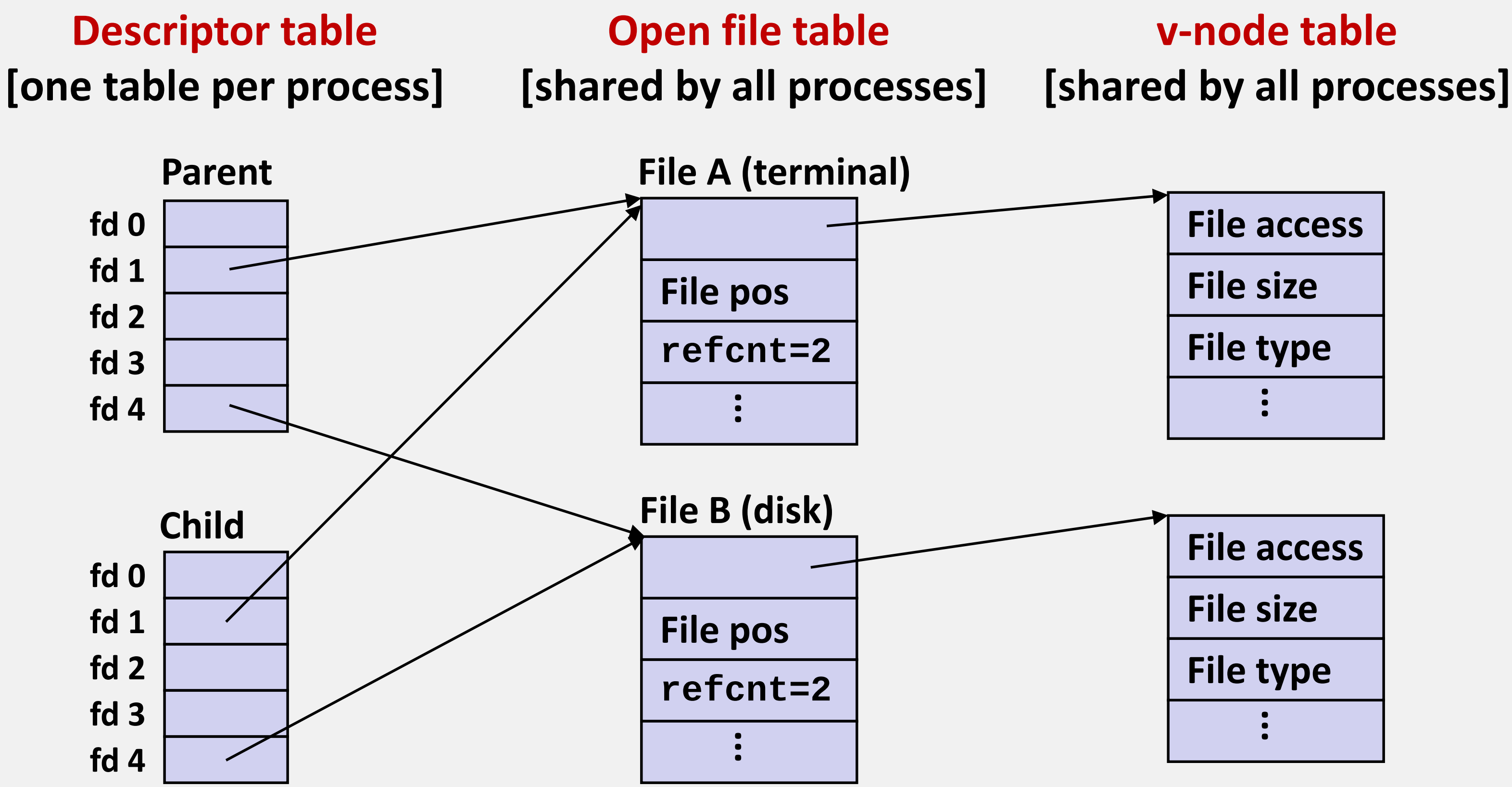
How Processes Share Files: `fork`

- **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)`**

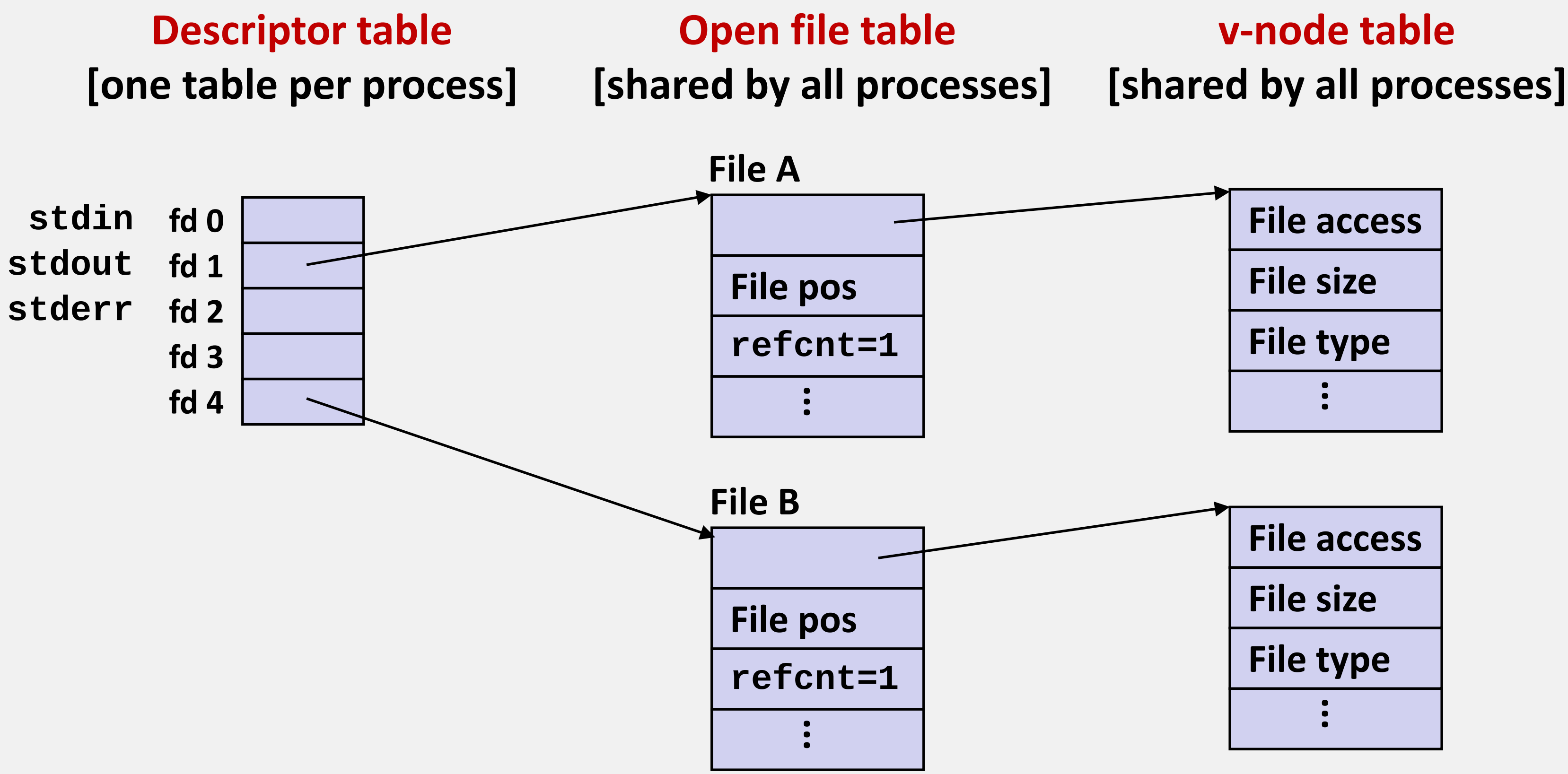
fd 0	
fd 1	a
fd 2	
fd 3	
fd 4	b

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

fd 0	
fd 1	b
fd 2	
fd 3	
fd 4	b

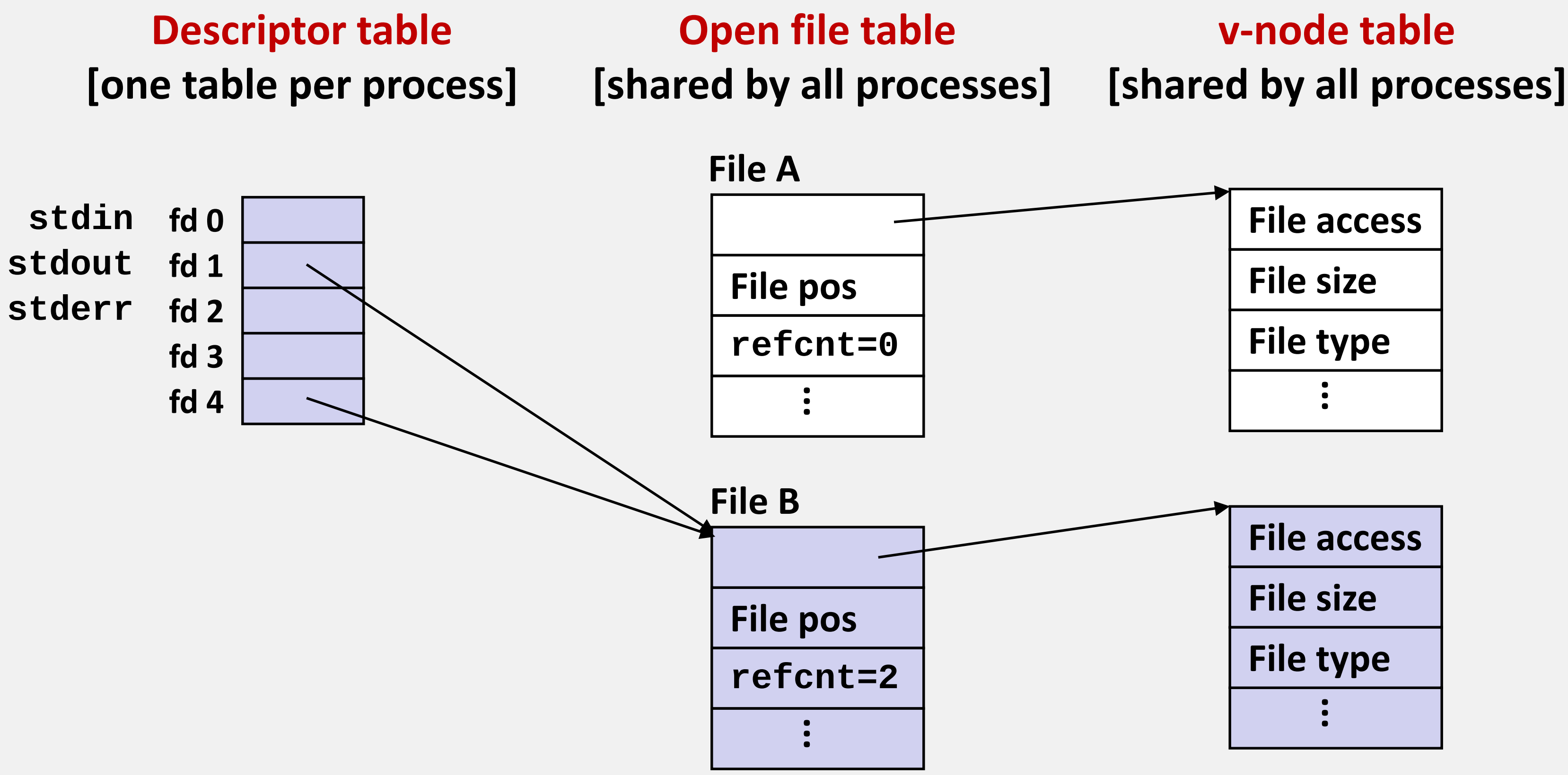
I/O Redirection Example

- **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



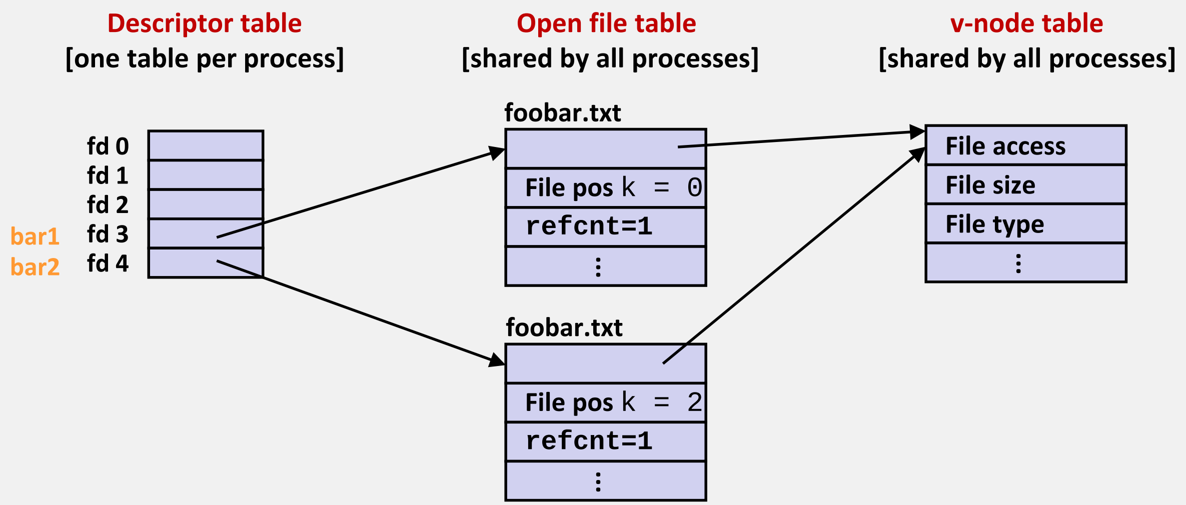
I/O Redirection Example 2:

```
linux> cat foobar.txt
CSE320
linux>
```

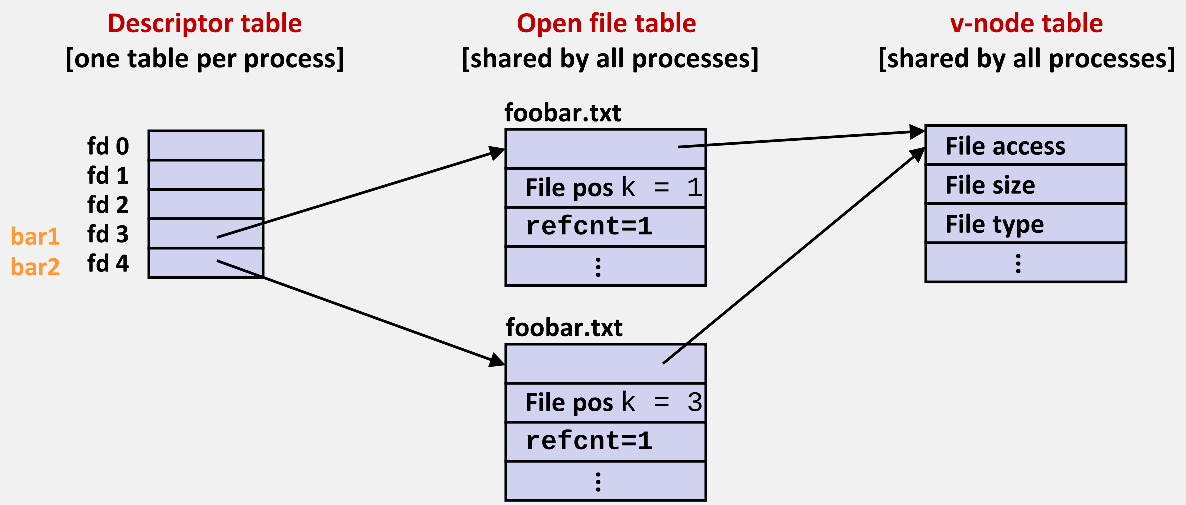
```
int main (int argc, char **argv)
{
    int bar1, bar2;
    char c;

    bar1 = Open(“foobar.txt”, O_RDONLY, 0);
    bar2 = Open(“foobar.txt”, O_RDONLY, 0);

    Read(bar2, &c, 1);   /* c = ‘C’; file position moves 1 */
    Read(bar2, &c, 1);   /* c = ‘S’; file position moves 1 */

    /* BREAK 1 */

    Read(bar1, &c, 1);   /* c = ‘C’; file position moves 1 */
    Read(bar2, &c, 1);   /* c = ‘E’; file position moves 1 */

    /* BREAK 2 */

    printf(“c = %c\n", c);
    exit(0);
}
```

I/O Redirection Example 2 (cont.)

- At /* BREAK 1*/



- **At /* BREAK 2*/**



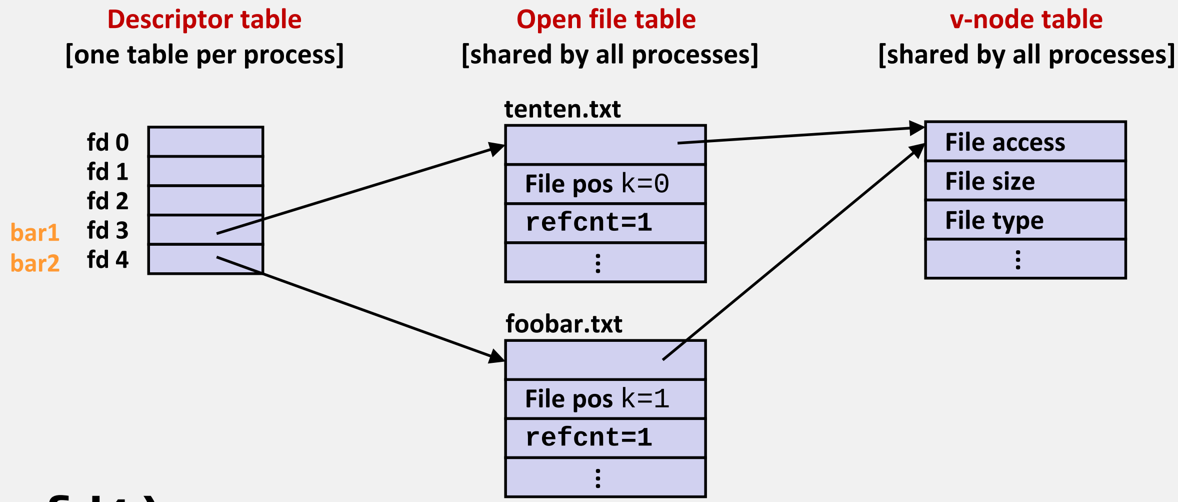
I/O Redirection Example 3:

```
linux> cat foobar.txt
CSE320
linux> cat tenten.data
1010
```

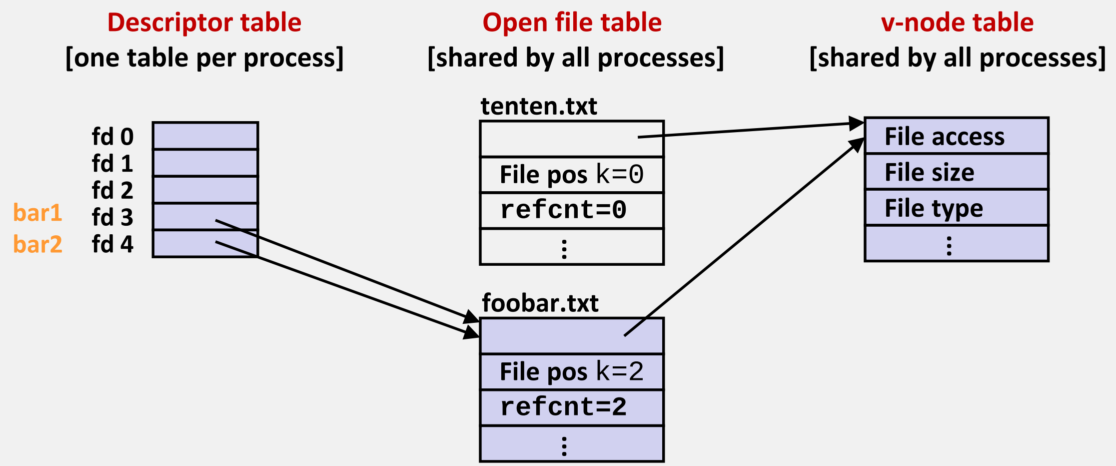
```
int main (int argc, char **argv)
{
    int bar1, bar2;
    char c;

    bar1 = Open("foobar.txt", O_RDONLY, 0);
    bar2 = Open("tenten.data", O_RDONLY, 0);

    Read(bar2, &c, 1);   /* c = 'C'; file position moves 1 */
    Dup2(bar2, bar1);
    Read(bar1, &c, 1);   /* c = 'S'; file position moves 1 */

    printf("c = %c\n", c);
    exit(0);
}
```

I/O Redirection Example 3 (cont.)

- Before `Dup2(fd2, fd1);`



- **After `Dup2(fd2, fd1);`**

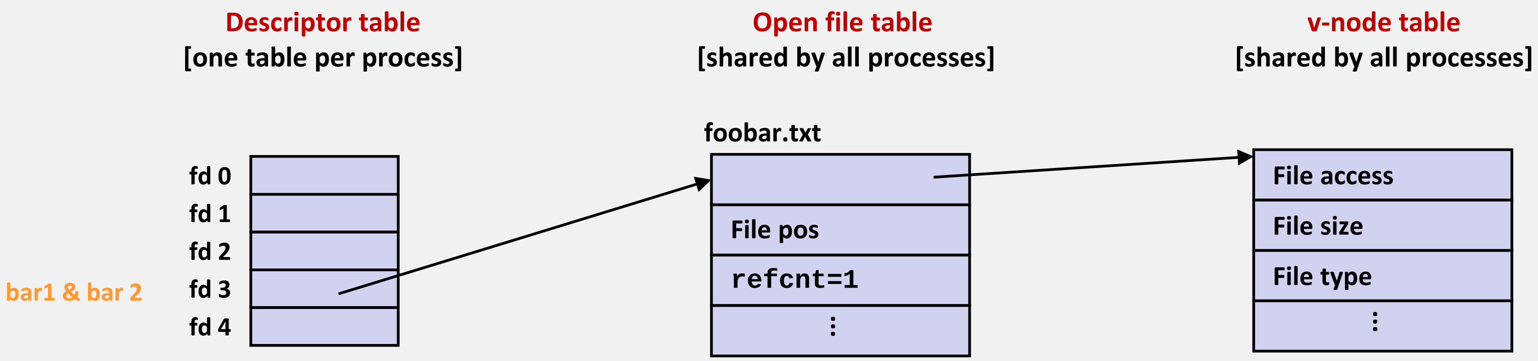


I/O Redirection Example 4:

```
linux> cat foobar.txt
CSE320
linux>
```

```
int main (int argc, char **argv)
{
    int bar1, bar2;
    char c;

    bar1 = Open(“foobar.txt”, O_RDONLY, 0);
    bar2 = bar1;

    Read(bar2, &c, 1);   /* c = ‘C’; file position moves 1 */
    Read(bar2, &c, 1);   /* c = ‘S’; file position moves 1 */
    Read(bar1, &c, 1);   /* c = ‘E’; file position moves 1 */

    printf(“c = %c\n", c);
    exit(0);
}
```

I/O Redirection Example 4 (cont.)



Difference Between Redirection & Pipes

`linux> ls > foo.txt` **VS.** `linux> ls | grep "a"`

- **Redirection '>' is used for passing output to either a *file* or *stream***
- **Pipe '|' is used to pass output to another *program* or *utility*.**
  - `linux> ls | grep "a"` **is functionally equivalent to**

    `linux> ls  > temp_file && grep "a" < temp_file`
  - **Pipes were created to simplify this procedure**
    - `temp_file` **does not need to be created & deleted with pipes (no disk I/O!)**
    - **Argument after '>' is expected to be a file (will not try to execute)**
    - **Argument after '|' is expected to be an executable (will not create file)**

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

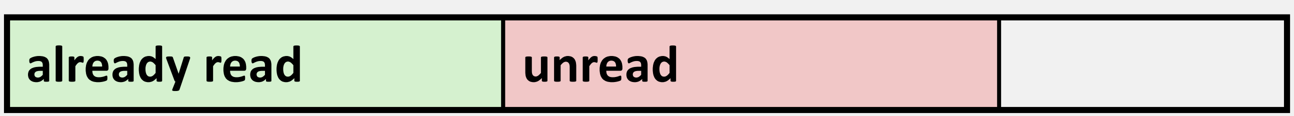
- **C programs begin life with three open streams (defined in `stdio.h`)**
  - **`stdin`** (standard input)
  - **`stdout`** (standard output)
  - **`stderr`** (standard error)

```
#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

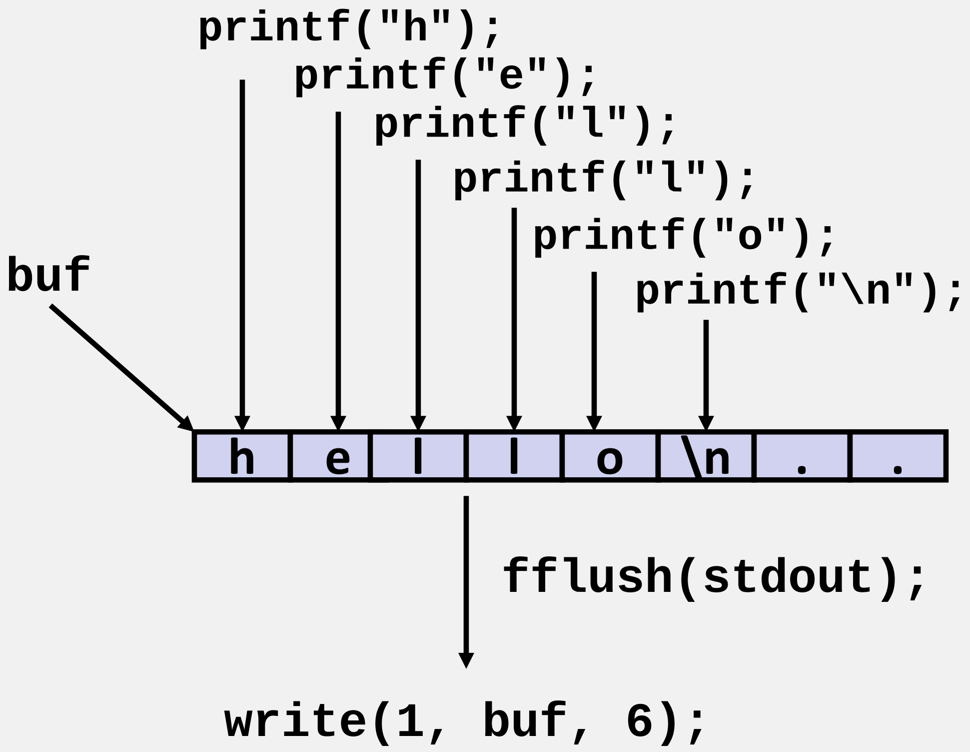
- **Applications often read/write one character at a time**
  - `getc, putc, ungetc`
  - `gets, fgets`
    - Read line of text one character at a time, stopping at newline
- **Implementing as Unix I/O calls expensive**
  - `read` and `write` require Unix kernel calls
    - > 10,000 clock cycles
- **Solution: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

Buffer



Buffering in Standard I/O

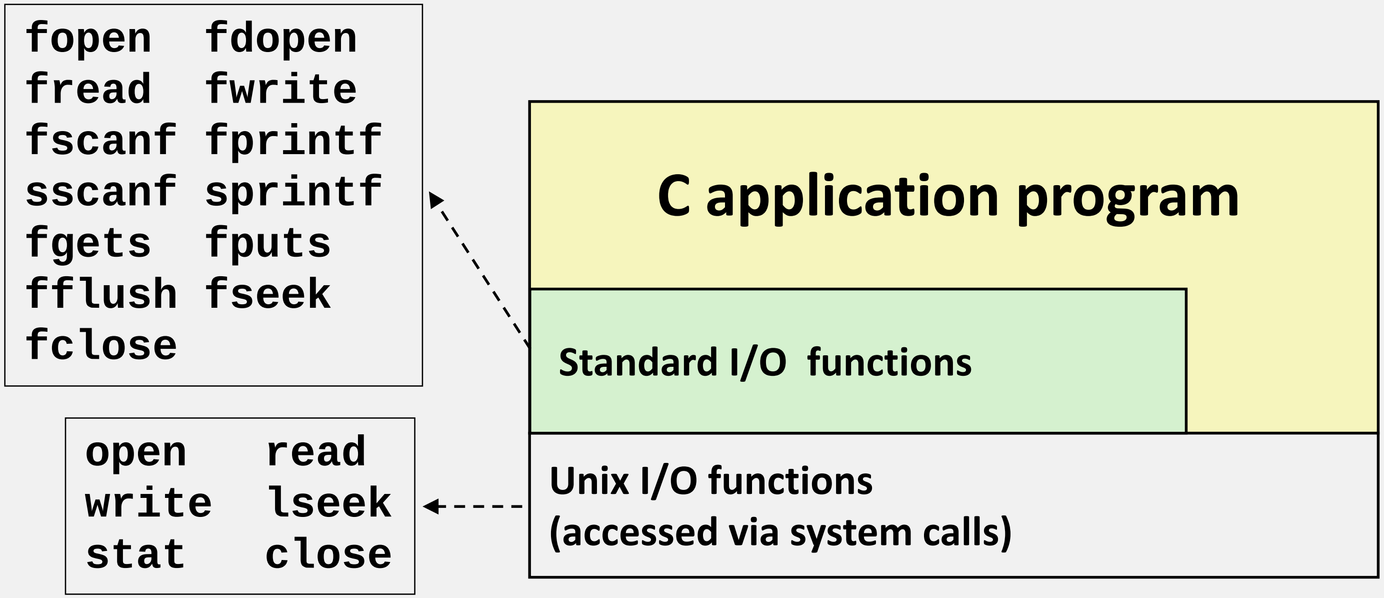
- Standard I/O functions use buffered I/O



- Buffer flushed to output fd on "\n", call to `fflush` or `exit`, or return from `main`.

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
  - 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
  - Both of these issues are addressed by the standard I/O package

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)

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
  - 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

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's 1993 classic text

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