CSCI2467: Systems Programming Concepts

Slideset 8: System System Level I/O

Source: CS:APP Chapter 10, Bryant & O'Hallaron

Instructor: M. Toups

Spring 2020

THE UNIVERSITY of NEW ORLEANS

DEPARTMENT OF COMPUTER SCIENCE

Shell lab: testing/grading

- I/O redirection will be covered today

Shell lab: testing/grading

- Use `make test01` etc
- **and** use `make rtest01` etc
- Compare behavior *and* output.
- Output must match!
- When you've started passing more tests, use `checktsh.py` to run all tests
- The **Checking Your Work** section of the shell lab writeup has more, use it

Shell lab
jobs list

- `jobs` is a global variable (array of `job_t` structs), accessible throughout the `tsh` program.
  - already created at top of `tsh.c`
  - managed by `addjob() and deletejob()`
  - These helper functions are given to you, but you must use them correctly!
- book's use of `addjob` in Figure 8.39 (p. 777) is intentionally flawed
  - avoid the race condition! (more shortly)
  - find a better way to block signals to ensure proper ordering

Shell lab
Hints

- the name `waitpid` can be misleading
  - it does more than just wait, also reaps
- You should use `waitpid` in your `sigchld_handler`
- `waitfg` can be pretty simple
  - just checks the jobs list repeatedly
  - returns once there is no more foreground job
  - but it only works if your jobs list is correctly updated!
  - (which depends on signal handlers)
- You may want to use `fgpid()` and/or `getjobpid()`

  putting a while loop around `sleep(1)` is fine
- Much more in the writeup's **Hints** section - use it!

Today

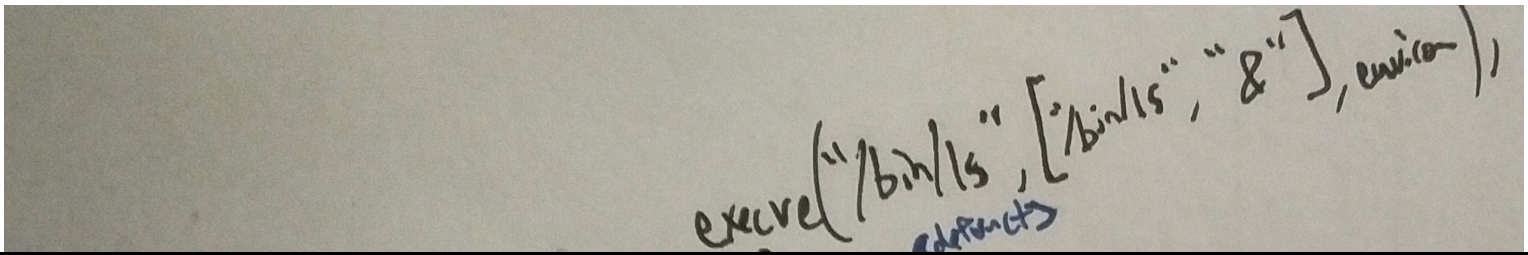
1. Shell lab note
1. Race conditions
2. System Level I/O
   - Unix I/O
   - Files
   - Opening and closing files
   - Metadata, sharing, and redirection
   - Standard I/O Functions
   - Applying what we know to Shell lab

Race condition

What is a *race condition*? See CS:APP3e section **8.5.6 Synchronizing Flows to Avoid Nasty Concurrency Bugs**

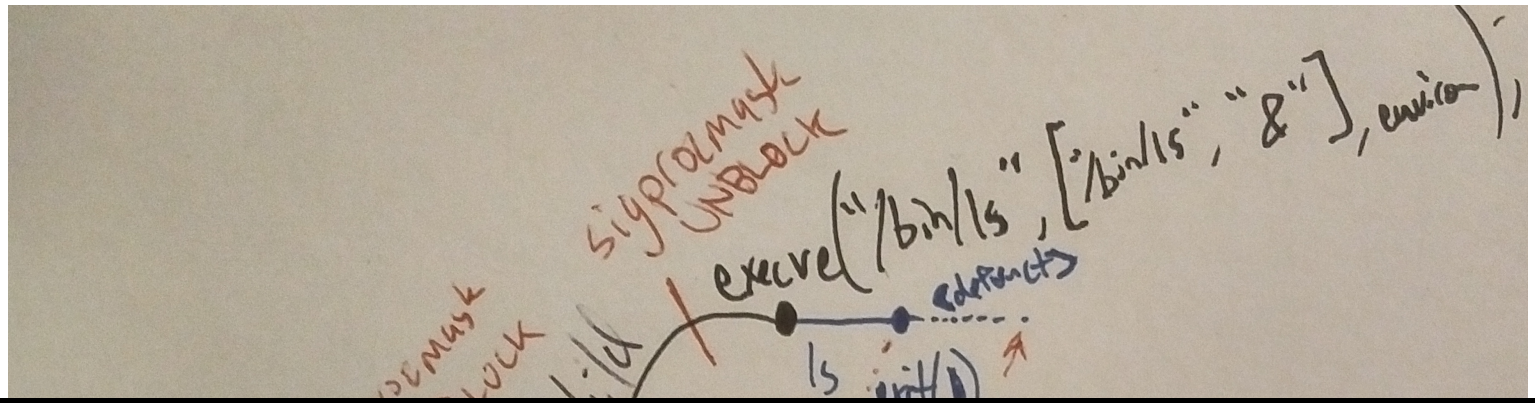
Race condition in `tsh`

- Recall: parent and child processes may run in unpredictable order (OS scheduler, etc)
- Seen in `forks.c` examples, also in `forkSig.c`
- So we have no way to know if parent or child goes first after `fork()`
- If child process is short-lived (such as `/bin/ls` which completes almost immediately) we have a problem:
- sometimes child will finish and generate `SIGCHLD` before parent has reached `addjob()`
- if `sigchld_handler` reaps and calls `deletejob()` (as it should), then the job will be deleted before it is added
- consequence: `addjob()` will come after `deletejob()` so that job will never be deleted



Solving race condition in `tsh`

- Solution for this race condition: signal blocking
- We can temporarily block signals using `sigprocmask()`
  - do this before `fork()`
  - be aware: signal mask is inherited by child process! (just like everything else)
  - child must unblock signals before `execve()`
  - parent does not unblock signals until after `addjob()`
- any signals that arrived while signals are blocked would be *pending*
- upon unblock, signal handler will run as usual
- signal is only blocked temporarily (postponed)



Today

0. Shell lab note
1. Race conditions
2. System Level I/O
   - Unix I/O
   - Files
   - Opening and closing files
   - Metadata, sharing, and redirection
   - Standard I/O Functions
   - Applying what we know to Shell lab

What is a file?

- Computer: stores information

- Files contain data

- Files have different formats

- Files are organized in some way

Overview of files in modern operating systems

- **A Linux *file* is a sequence of *m* bytes:**
  - $B_0, B_1, \ldots, B_k, \ldots, B_{m-1}$
- **A Linux *file* is a sequence of *m* bytes:**
  - $B_0, B_1, \ldots, B_k, \ldots, B_{m-1}$
- **Cool fact: All I/O devices are represented as files:**
  - **`/dev/sda2`** (**`/usr`** disk partition)
  - **`/dev/tty2`** (terminal)

Overview of files in Unix

- **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()`**
- **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 next offset into file to read or write

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
- **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`')
    - Newline is `0xa`, same as ASCII line feed character (LF)

- **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***

Directories

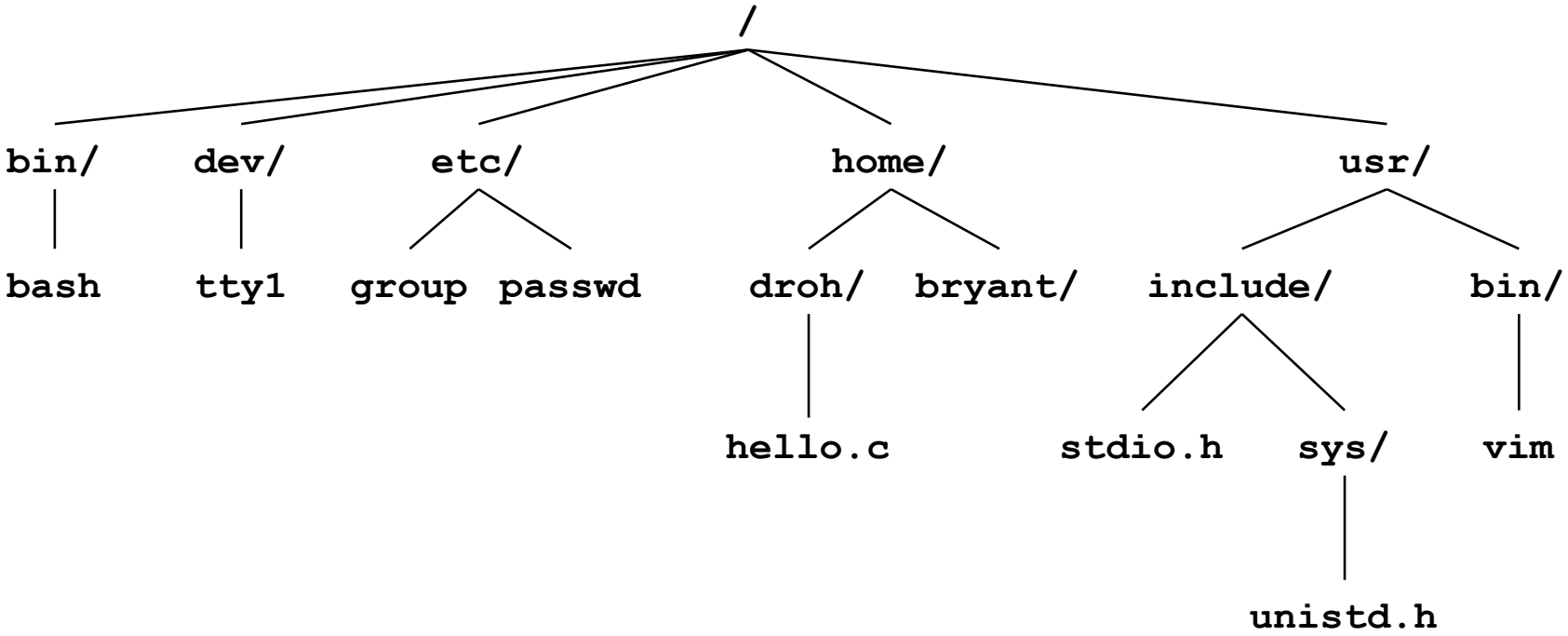
- **Directory consists of an array of *links***
  - Each link maps a *filename* to a file
- **Each directory contains at least two entries**
  - `.` (dot) is a link to itself
  - `..` (dot dot) is a link to *the parent directory* in the *directory hierarchy* (next slide)

- **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

Directory Hierarchy

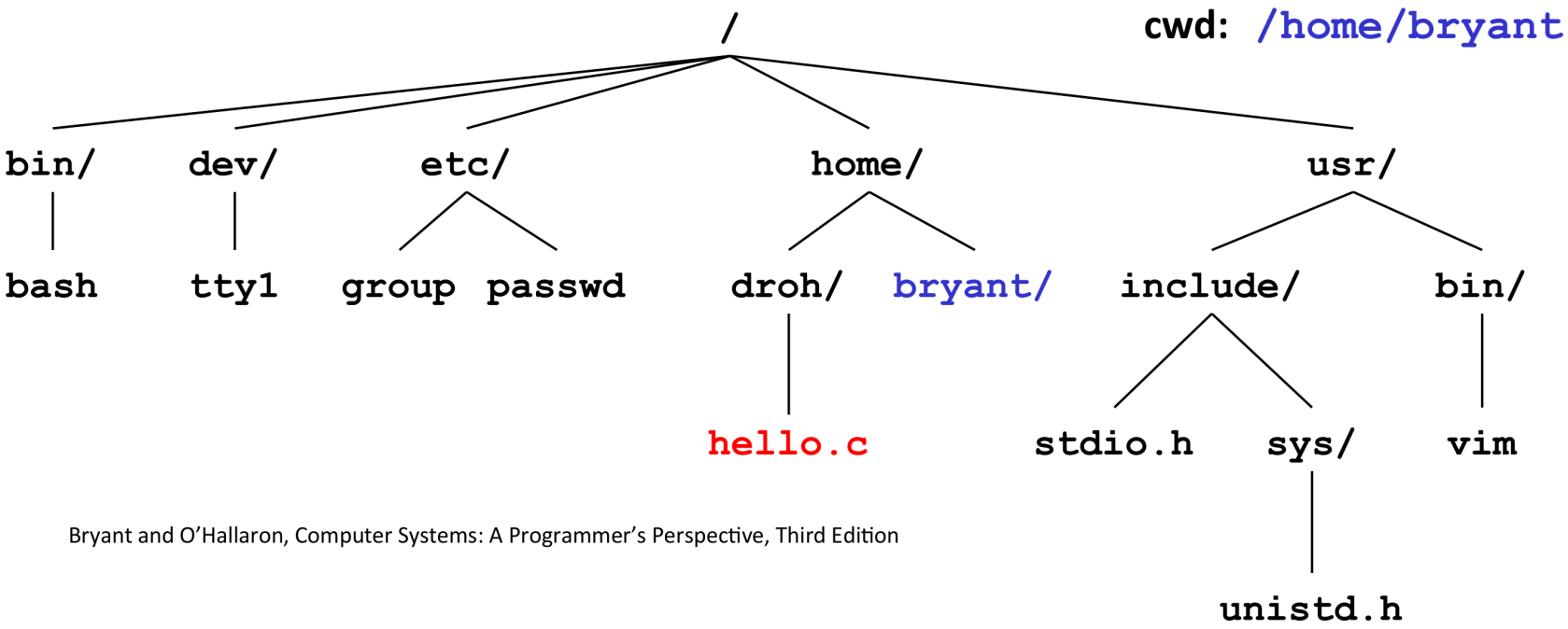
- **All files are organized as a hierarchy anchored by root directory named / (slash)**



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

Path names

- **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`



Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

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:**
  - 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 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 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);
}
```

Today

0. Shell lab note
1. Race conditions
2. System Level I/O
   - Unix I/O
   - Files
   - Opening and closing files
   - Metadata, sharing, and redirection
   - Standard I/O Functions
   - Applying what we know to Shell lab

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

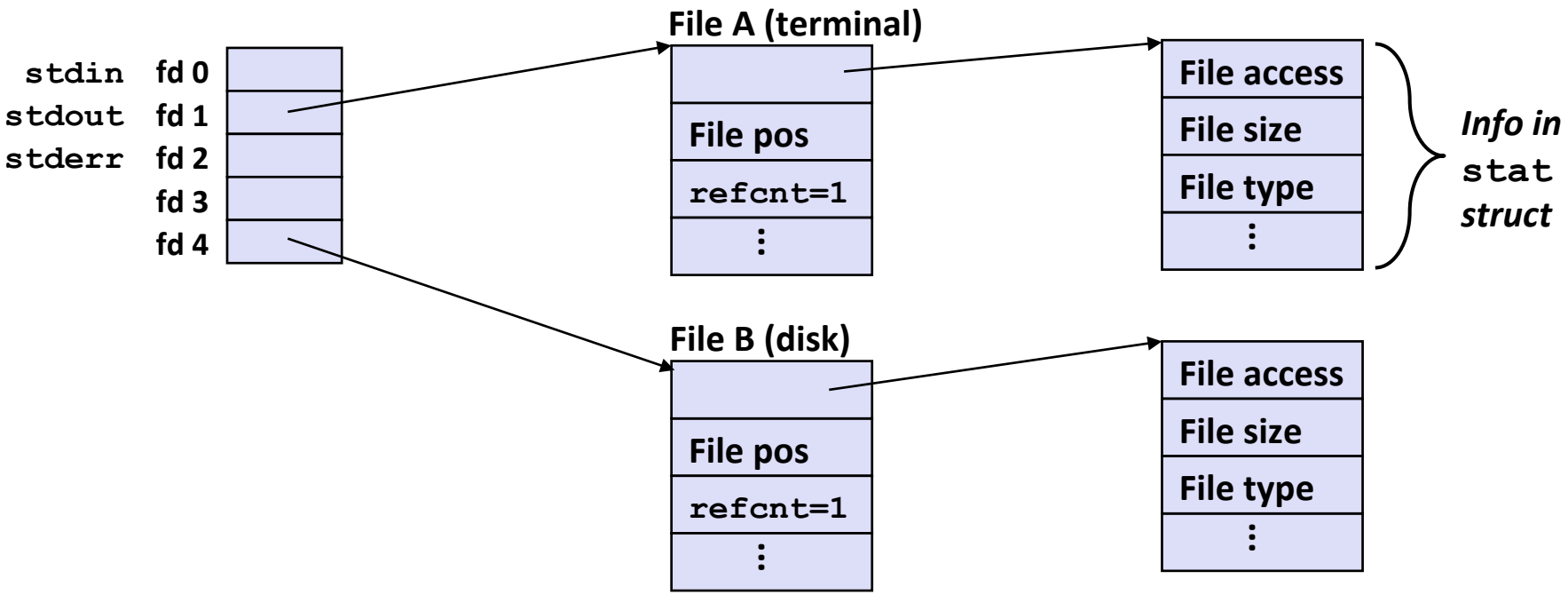
```
int main (int argc, char **argv)
{
    struct stat stat;
    char *type, *readok;

    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
```

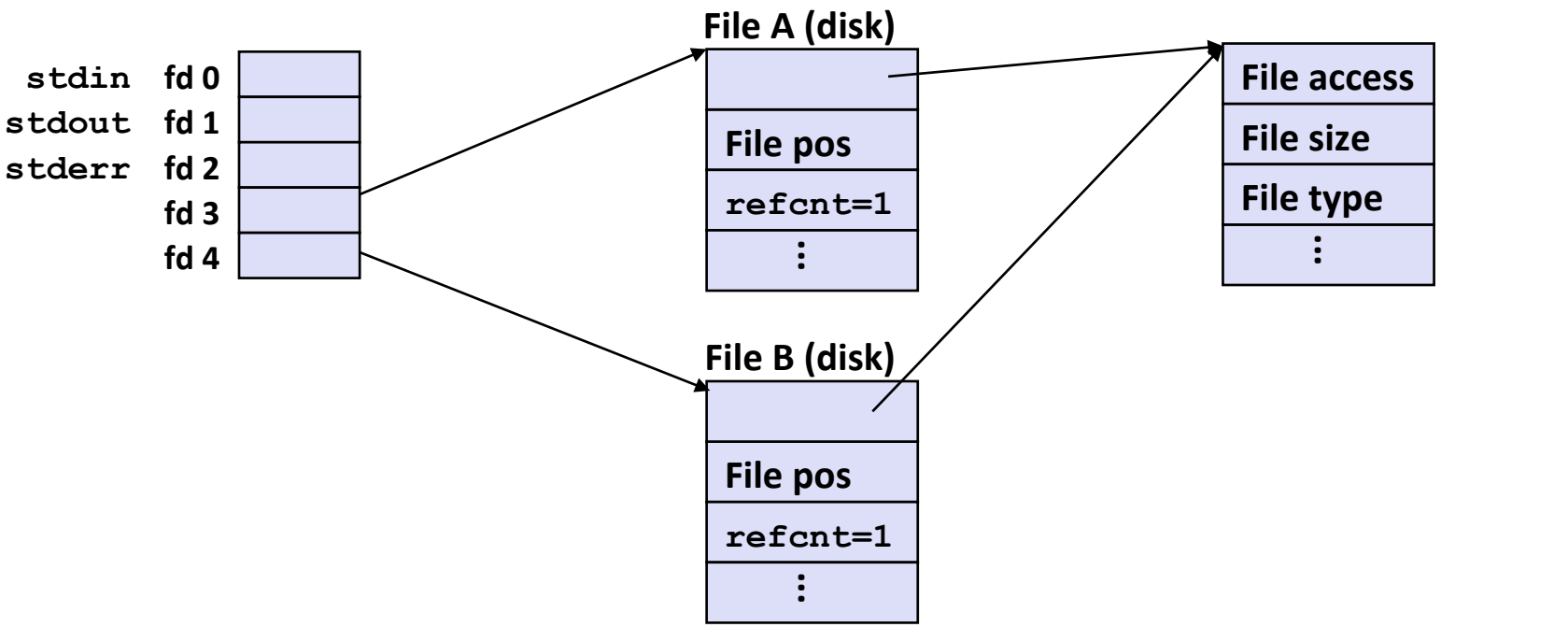
How the Unix kernel represents open files

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



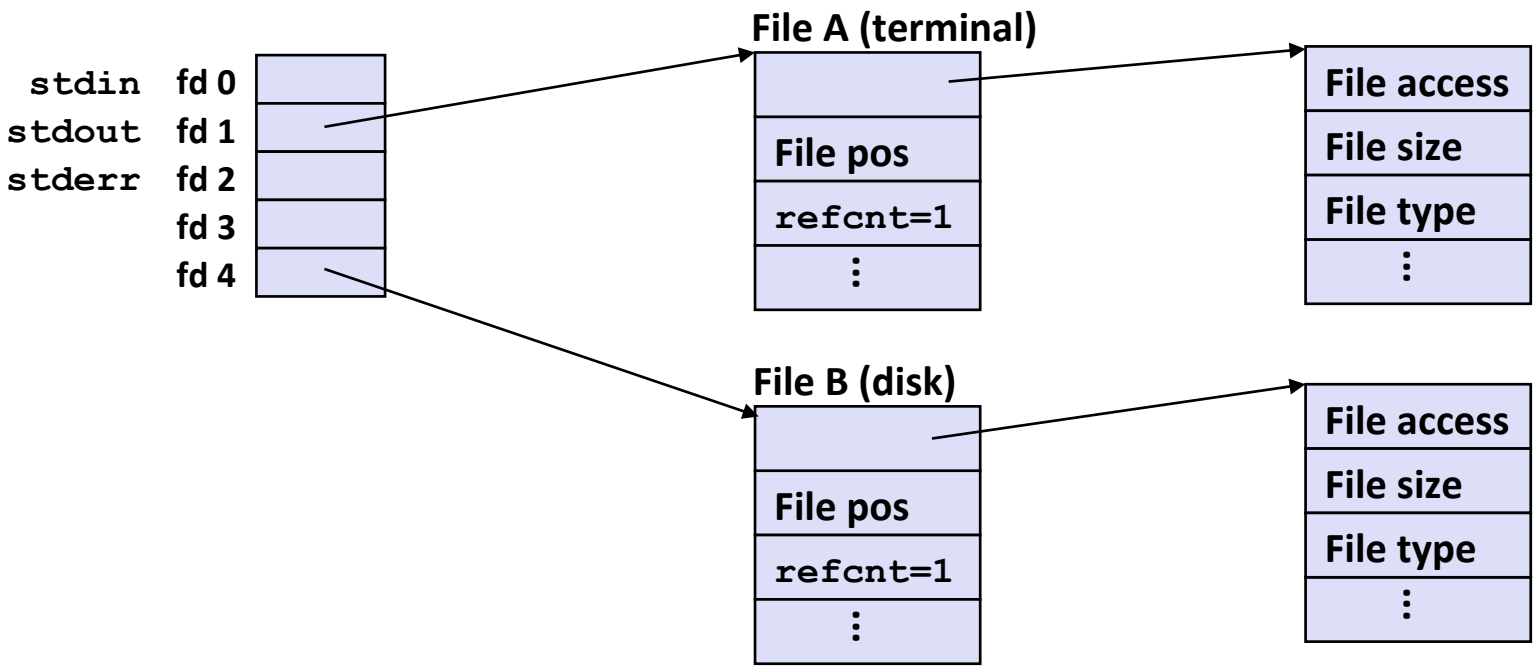
File sharing

- **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:**



- **A child process inherits its parent's open files**

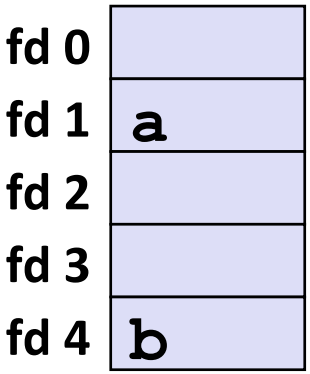
I/O redirection

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

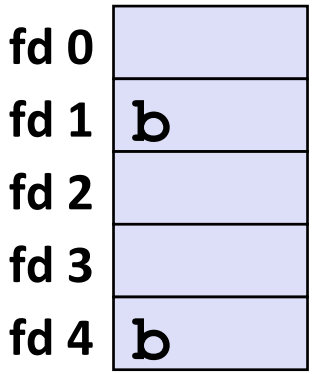
```
linux> ls > foo.txt
```

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

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

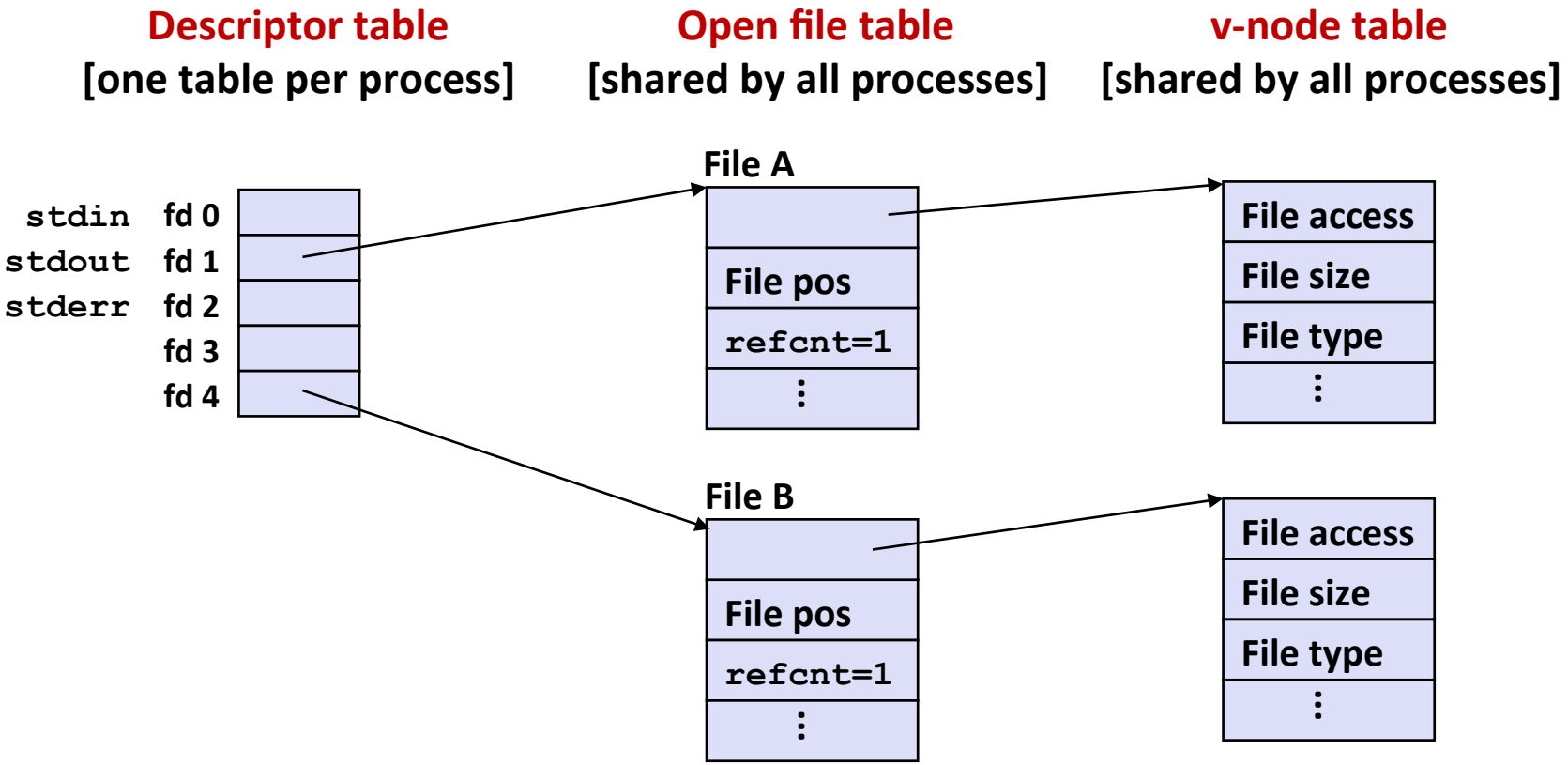


Descriptor table *after* `dup2(4,1)`



I/O redirection example

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



- **Step #2: call dup2(4, 1)**

Today

- Shell lab note
- Race conditions
- System Level I/O
  - Unix I/O
  - Files
  - Opening and closing files
  - Metadata, sharing, and redirection
  - Standard I/O Functions
  - Applying what we know to Shell lab

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

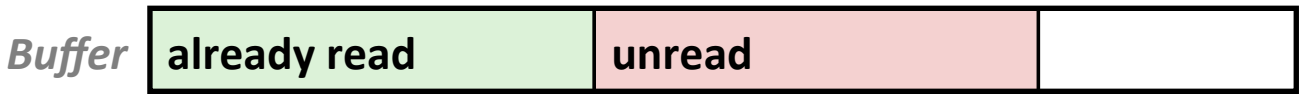
- **Standard I/O models open files as *streams***
  - Abstraction for a file descriptor and a buffer in memory

- **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");
}
```

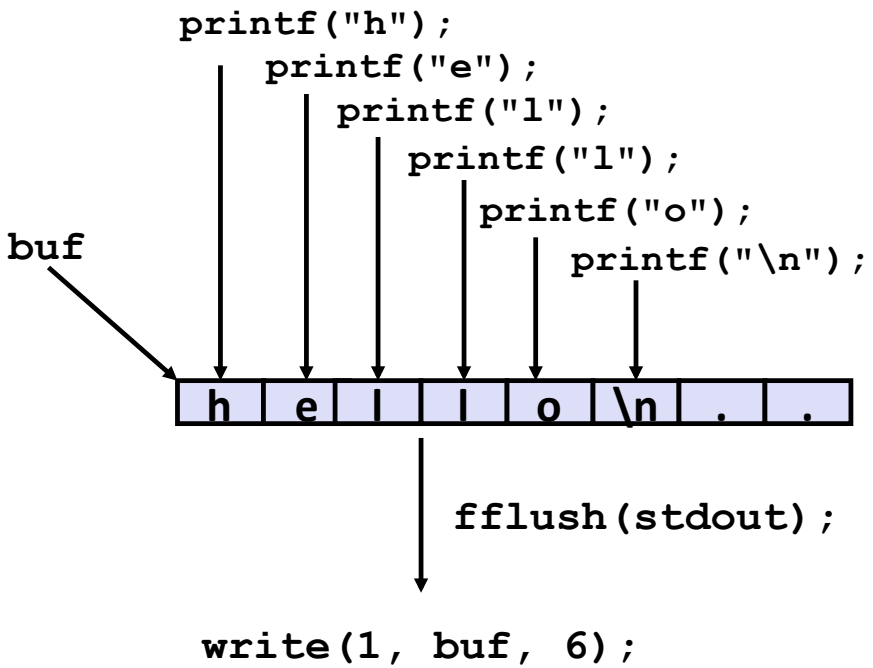
Buffered I/O: Motivation

- **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)                           = ?
```

Today

0. Shell lab note
1. Race conditions
2. System Level I/O
   - Unix I/O
   - Files
   - Opening and closing files
   - Metadata, sharing, and redirection
   - Standard I/O Functions
   - Applying what we know to Shell lab

Redirecting I/O

Now, let's apply this in shell lab...

- `do_redirect(argv)` function is started for you!
  - takes `argv` as input, will detect use of > and < for you
  - your job: given a filename, open a file with that name, use `dup2()` to change either standard input or standart output to the new file descriptor
  - after that you can simply call `close()` and you're done
- test cases:

```
$ /bin/ls -l tsh.c > testout
```

- Does this write the output (stdout) of `ls` into `testout`?

```
$ ./myread 40 < tsh.c
```

- Does it read input from `tsh.c` and use that as `stdin`?

Shell lab
Input/output redirection

- You need to use:
  - `open()` and `close()` (see CS:APP3e 10.3 for usage)
    (`man 2 open` in linux will give you even more detail)
    (no read() or write() - that will happen later, after `execve`)
  - `dup2()` (see 10.9 for usage)
- And tiny bit more:
  - declare an int to hold a file descriptor, use the correct entry in `argv`
  - check system calls for errors! (`return <0`)
    (file I/O may fail for many reasons, some are common, you need to handle that)

```
tsh> ./myread < dasf
open() error:  No such file or directory
```