15-213 Recitation 11

Shell lab: Processes, Signals, IO

April 4, 2022

Your TAs

Outline

- Logistics
- Process Lifecycle
- Signal Handling
- IO and File Descriptors

Learning Objectives

- **Expectations:**
  - Basic understanding of signals & processes
- **Goals:**
  - Better understanding of signals & processes

Logistics

- Shell Lab due Apr 14th
  - Code Review Signup due Apr 14th
  - Check Website for updated code review signups deadlines in the future

Shell Lab

- **Due date:** Apr 14th
- Simulate a Linux-like shell
- **Review the write-up carefully.**
  - Review once before starting, and again when halfway through
  - This will save you a lot of style points and a lot of grief!
- **Read Chapter 8 in the textbook:**
  - Process lifecycle and signal handling
  - How race conditions occur, and how to avoid them
  - **Be careful not to use code from the textbook without understanding it first.**

Process Graphs

- **How many different lines could be printed?**

```
int main(void) {
    char *tgt = "child";
    sigset_t mask, old_mask;
    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);
    sigprocmask(SIG_SETMASK, &mask, &old_mask); // Block
    pid_t pid = fork();
    if (pid == 0) {
        pid = getppid(); // Get parent pid
        tgt = "parent";
    }
    kill(pid, SIGINT);
    sigprocmask(SIG_SETMASK, &old_mask, NULL); // Unblock
    printf("Sent SIGINT to %s:%d\n", tgt, pid);
    exit(0);
}
```

Process Graphs

- **How many different lines are printed?**

```
int main(void) {
    char *tgt = "child";
    sigset_t mask, old_mask;
    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);
    sigprocmask(SIG_SETMASK, &mask, &old_mask); // Block
    pid_t pid = fork();
    if (pid == 0) {
        pid = getppid(); // Get parent pid
        tgt = "parent";
    }
    kill(pid, SIGINT);
    sigprocmask(SIG_SETMASK, &old_mask, NULL); // Unblock
    printf("Sent SIGINT to %s:%d\n", tgt, pid);
    exit(0);
}
```

0 or 1 line. The parent and child try to terminate each other.

Signals and Handling

- **Signals can happen at any time**
  - Control when through blocking signals

- **Signals also communicate that events have occurred**
  - What event(s) correspond to each signal?

- **Write separate routines for receiving (i.e., signals)**

Counting with signals

- **Will this code terminate?**

```
volatile int counter = 0;
void handler(int sig) { counter++; }

int main(void) {
    signal(SIGCHLD, handler);
    for (int i = 0; i < 10; i++) {
        if (fork() == 0) { exit(0); }
    }
    while (counter < 10) {
        mine_bitcoin();
    }
    return 0;
}
```

Counting with signals (you can’t)

- **Will this code terminate?**

```
volatile int counter = 0;
void handler(int sig) { counter++; }

int main(void) {
    signal(SIGCHLD, handler);
    for (int i = 0; i < 10; i++) {
        if (fork() == 0) { exit(0); }
    }
    while (counter < 10) {
        mine_bitcoin();
    }
    return 0;
}
```

(Don't use `signal`, use `Signal` or `sigaction` instead!)

(Don't busy-wait, use `sigsuspend` instead!)

It might not, since signals can coalesce.

sigsuspend

```
int sigsuspend(const sigset_t *mask);
```

- Suspend current process until a signal is received, you can specify which one using a mask

This is an atomic version of:

```
sigprocmask(SIG_SETMASK, &mask, &prev)
pause();
sigprocmask(SIG_SETMASK, &prev, NULL);
```

- This still doesn’t fix the issue of signals coalescing!
- Don’t use pause() in your own code

Proper signal handling

- **How can we fix the previous code?**
  - Remember that signals will be coalesced, so the number of times a signal handler has executed is **not** necessarily the same as number of times a signal was sent.
  - We need some other way to count the number of children.

Proper signal handling

- **How can we fix the previous code?**
  - Remember that signals will be coalesced, so the number of times a signal handler has executed is **not** necessarily the same as number of times a signal was sent.
  - We need some other way to count the number of children.

```
void handler(int sig) {
    pid_t pid;
    while ((pid = waitpid(-1, NULL, WNOHANG)) > 0) {
        counter++;
    }
}
```

(This instruction isn't atomic. Why won't there be a race condition?)

Blocking signals

- **Surround blocks of code with calls to `sigprocmask`.**
  - Use SIG_BLOCK to block signals at the start.
  - Use SIG_SETMASK to restore the previous signal mask at the end.
- **Don't use SIG_UNBLOCK.**
  - We don't want to unblock a signal if it was already blocked.
  - This allows us to nest this procedure multiple times.

```
sigset_t mask, prev;
sigemptyset(&mask, SIGINT);
sigaddset(&mask, SIGINT);
sigprocmask(SIG_BLOCK, &mask, &prev);
// ...
sigprocmask(SIG_SETMASK, &prev, NULL);
```

Writing signal handlers

- **G1. Call only async-signal-safe functions in your handlers.**
  - Do not call `printf`, `sprintf`, `malloc`, `exit`! Doing so can cause deadlocks, since these functions may require global locks.
  - We've provided you with `sio_printf` which you can use instead.
- **G2. Save and restore `errno` on entry and exit.**
  - If not, the signal handler can corrupt code that tries to read `errno`.
  - The driver will print a warning if `errno` is corrupted.
- **G3. Temporarily block signals to protect shared data.**
  - This will prevent race conditions when writing to shared data.
- **Avoid the use of global variables in tshlab.**
  - They are a source of pernicious race conditions!
  - You do not need to declare any global variables to complete tshlab.
  - Use the functions provided by `tsh_helper`.

Error and signals : Recap

- **You can’t expect people to block signals around all error handling logic**
- **Hence, your signal handler shouldn’t interfere with them**
- **Solution:**
  - Do not make any system call that could set errno
  - Save and restore errno (store at beginning of handler and restore after)
  - Think about what would work for the case you are using, not one rule

IO functions

Needed for tshlab

- `int open(const char *pathname, int flags, mode_t mode);`
  - Can pass bitwise-or of flags:
    - File Creation: O_CREAT, O_TRUNC, etc.
    - Access Modes (must include one): O_RDONLY, O_WRONLY, O_RDWR
      - O_RDONLY|O_WRONLY doesn’t work! Use O_RDWR
  - Mode: specifies who else can read/write the new file
    - Required argument when O_CREAT is used
    - Use 0666 unless you have a specific reason to do something else

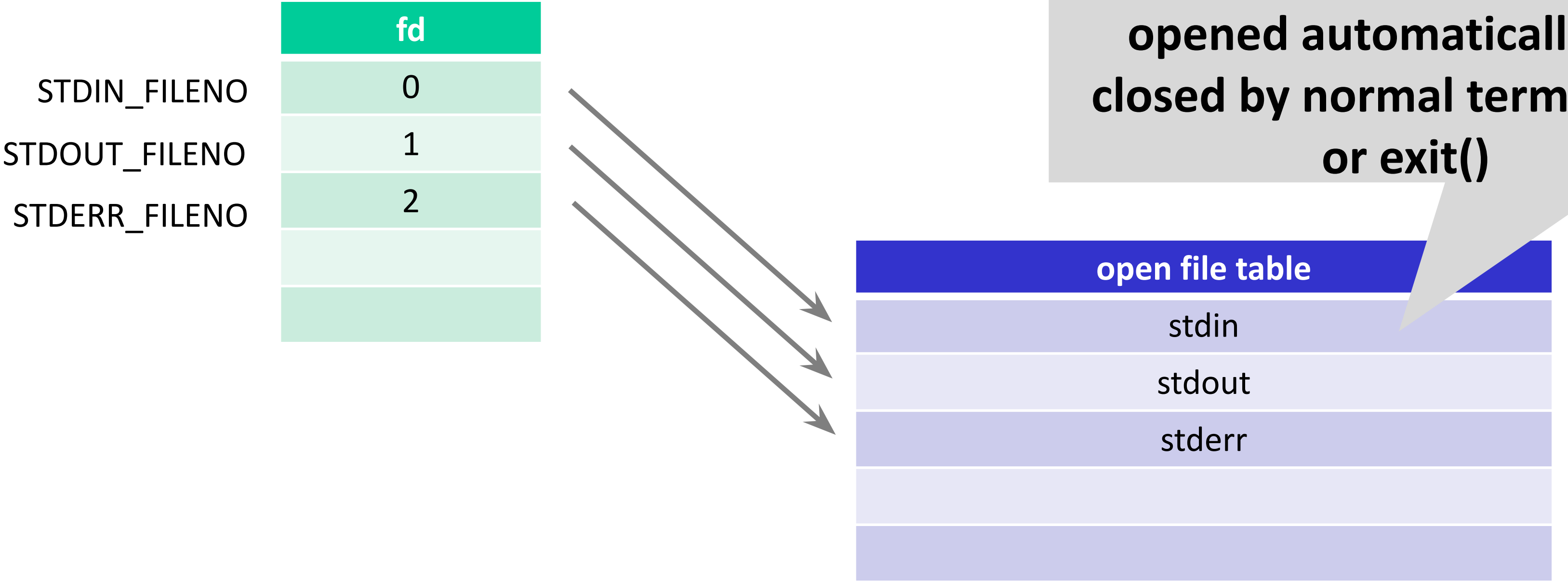
- `int close(int fd);`
- `int dup2(int oldfd, int newfd);`

Permissions for open()

	Read (R)	Write (W)	Executable (X)	All (RWX)
User (USR)	S_IRUSR	S_IWUSR	S_IXUSR	S_IRWXU
Group (GRP)	S_IRGRP	S_IWGRP	S_IXGRP	S_IRWXG
Other (OTH)	S_IROTH	S_IWOTH	S_IXOTH	S_IRWXO

- **These constants can be bitwise-OR'd and passed to the third argument of open()**
- **What does S_IRWXG | S_IXUSR | S_IXOTH mean?**
- **How to create a file which everyone can read from but only the user can write to it or execute it?**

STD File Descriptors

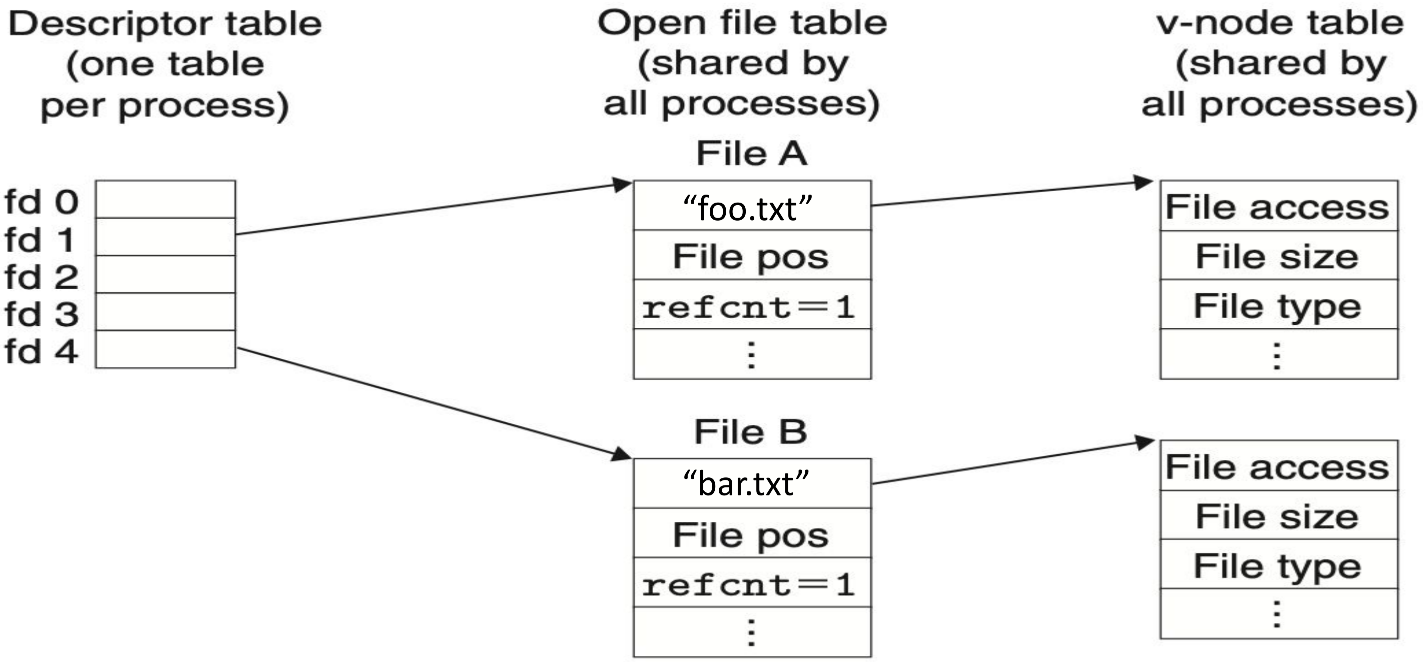


	fd
STDIN_FILENO	0
STDOUT_FILENO	1
STDERR_FILENO	2

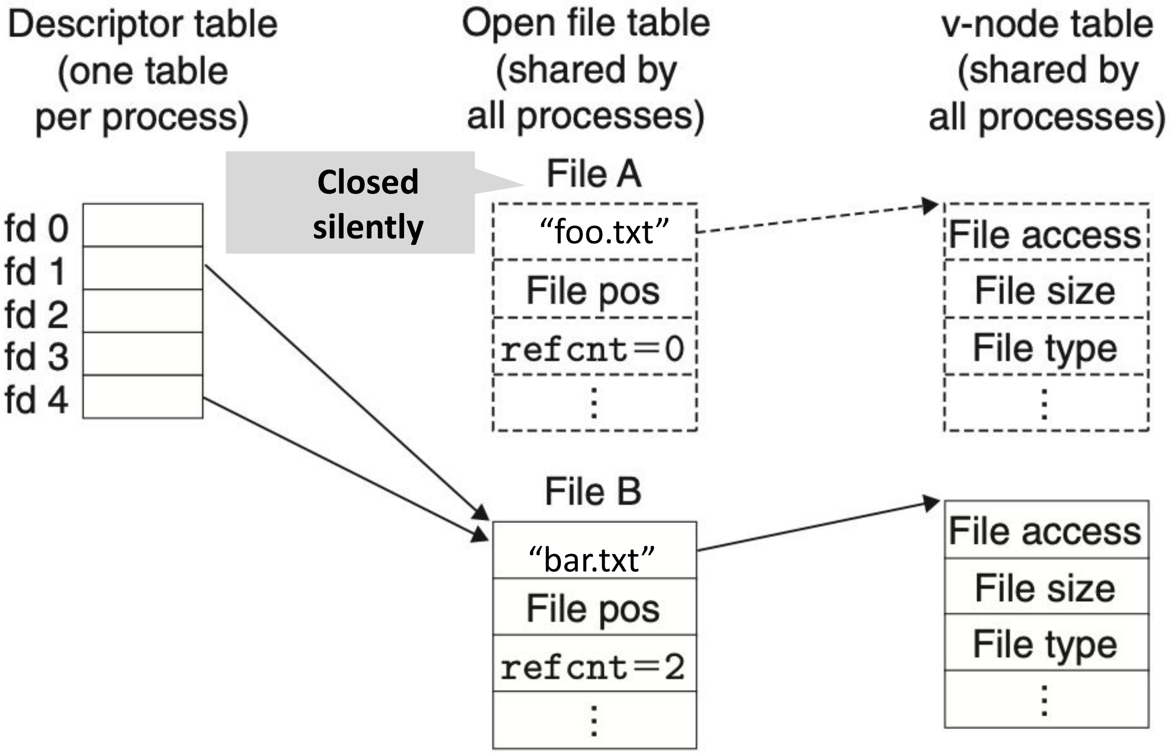
open file table
stdin
stdout
stderr

stdin, stdout, stderr are opened automatically and closed by normal termination or exit()

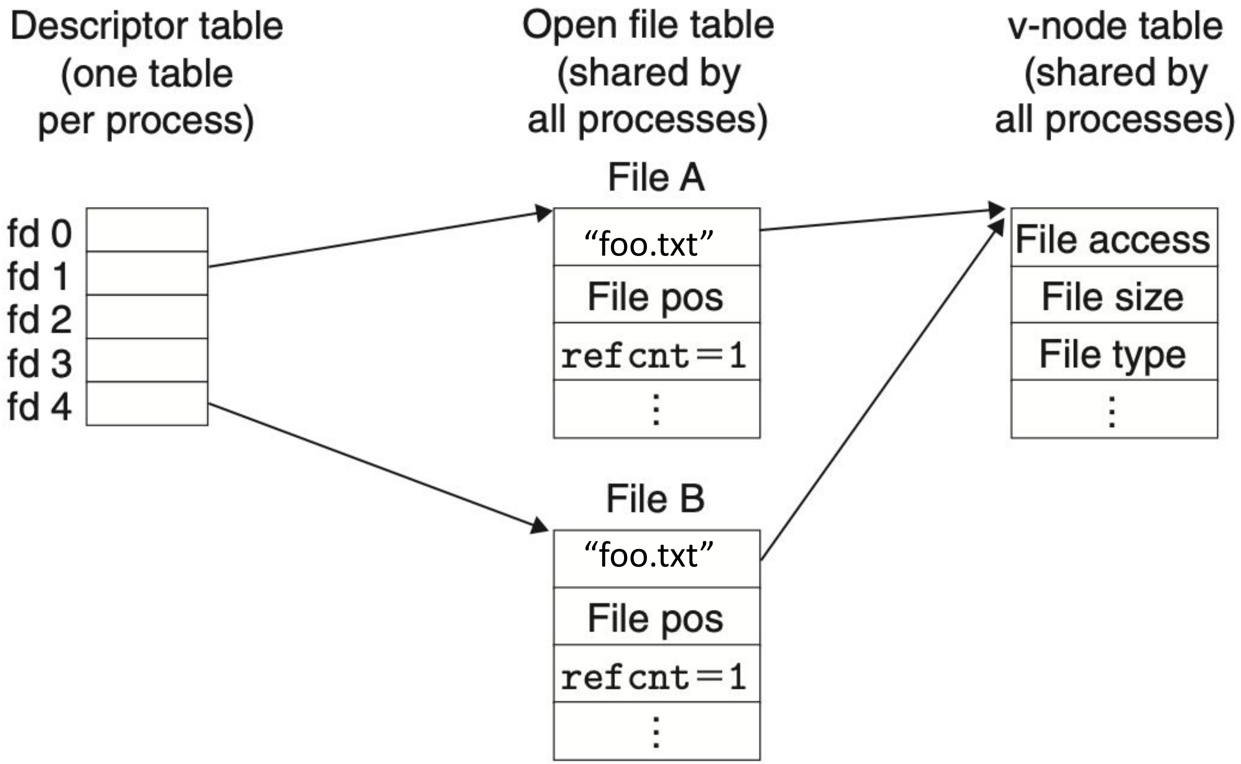
File descriptors (File A != File B)



File descriptors after dup2(4,1);

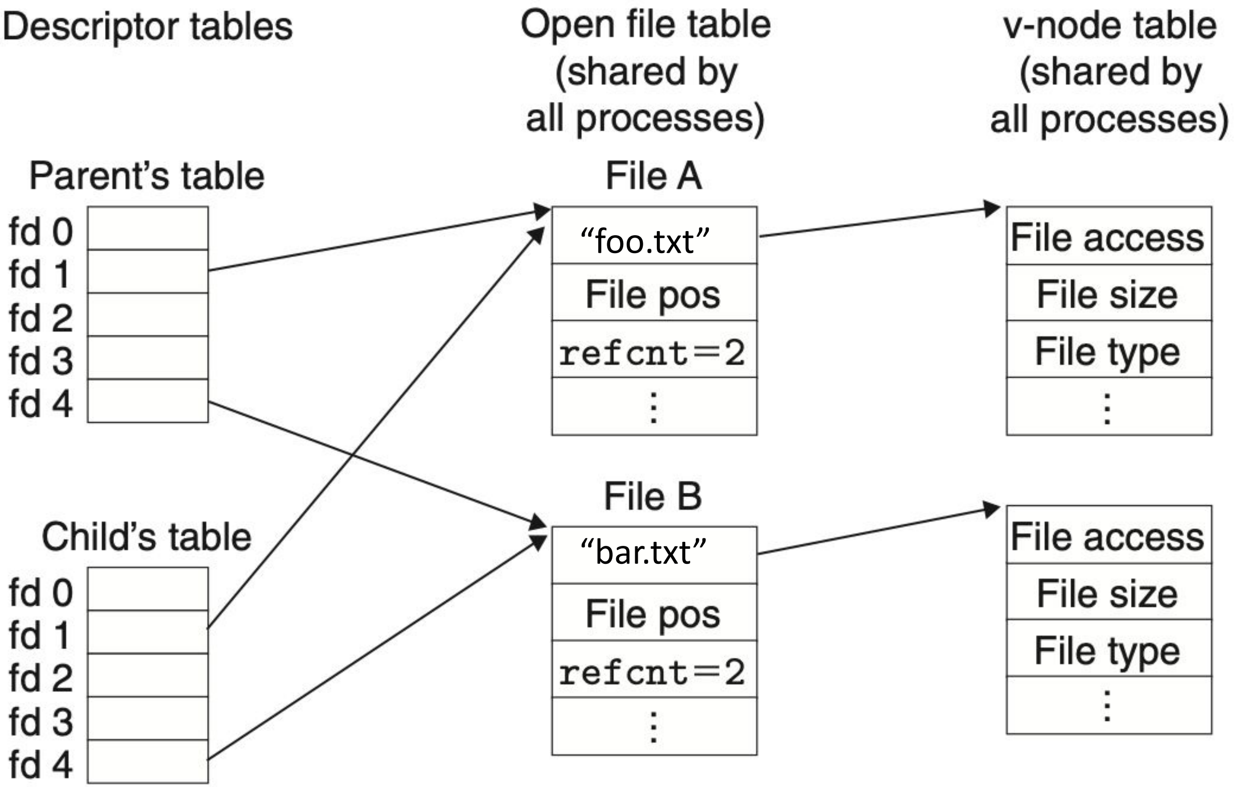


File Descriptors (File A == File B)



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

File Descriptors after a fork()



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

IO and Fork()

- File descriptor management can be tricky.
- How many file descriptors are open in the parent process at the indicated point?
- How many does each child have open at the call to execve?

```
int main(int argc, char** argv)
{
    int i;
    for (i = 0; i < 4; i++)
    {
        int fd = open(“foo”, O_RDONLY);
        pid_t pid = fork();
        if (pid == 0)
        {
            int ofd = open(“bar”, O_RDONLY);
            execve(...);
        }
    }
    // How many file descriptors are open in the parent?
```

Redirecting IO

- **At the two points (A and B) in main, how many file descriptors are open?**

```
int main(int argc, char** argv)
{
    int i, fd;
    fd = open(“foo”, O_WRONLY);
    dup2(fd, STDOUT_FILENO);
    // Point A
    close(fd);
    // Point B
    ...
```

Redirecting IO

- **File descriptors can be directed to identify different open files.**

```
int main(int argc, char** argv) {
    int i;
    for (i = 0; i < 4; i++)
    {
        int fd = open("foo", O_RDONLY);
        pid_t pid = fork();
        if (pid == 0)
        {
            int ofd = open("bar", O_WRONLY);
            dup2(fd, STDIN_FILENO);
            dup2(ofd, STDOUT_FILENO);
            execve(...);
        }
    }
    // How many file descriptors are open in the parent?
}
```

File IO Activity

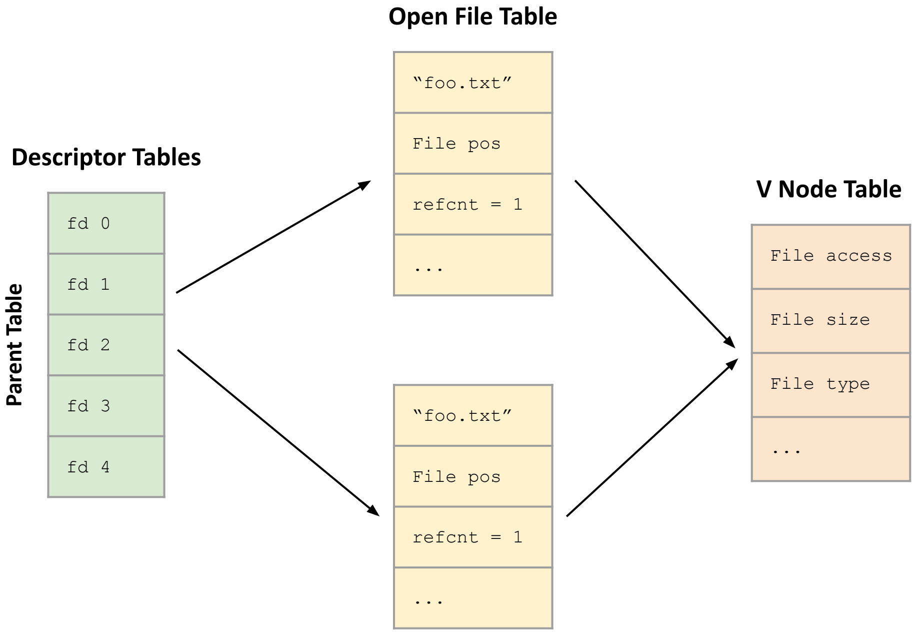
Activity Question

What is the possible output given contents of foo.txt are “ABCDEFG”?

```
int main(int argc, char *argv[]) {
        int fd1 = open("foo.txt", O_RDONLY);
        int fd2 = open("foo.txt", O_RDONLY);
        read_and_print_one(fd1);
        read_and_print_one(fd2);
        if(!fork()) {
                read_and_print_one(fd2);
                read_and_print_one(fd2);
                close(fd2);
                fd2 = dup(fd1);
                read_and_print_one(fd2);
        } else {
                wait(NULL);
                read_and_print_one(fd1);
                read_and_print_one(fd2);
                printf("\n");
        }
        close(fd1);
        close(fd2);
        return 0;
}
```

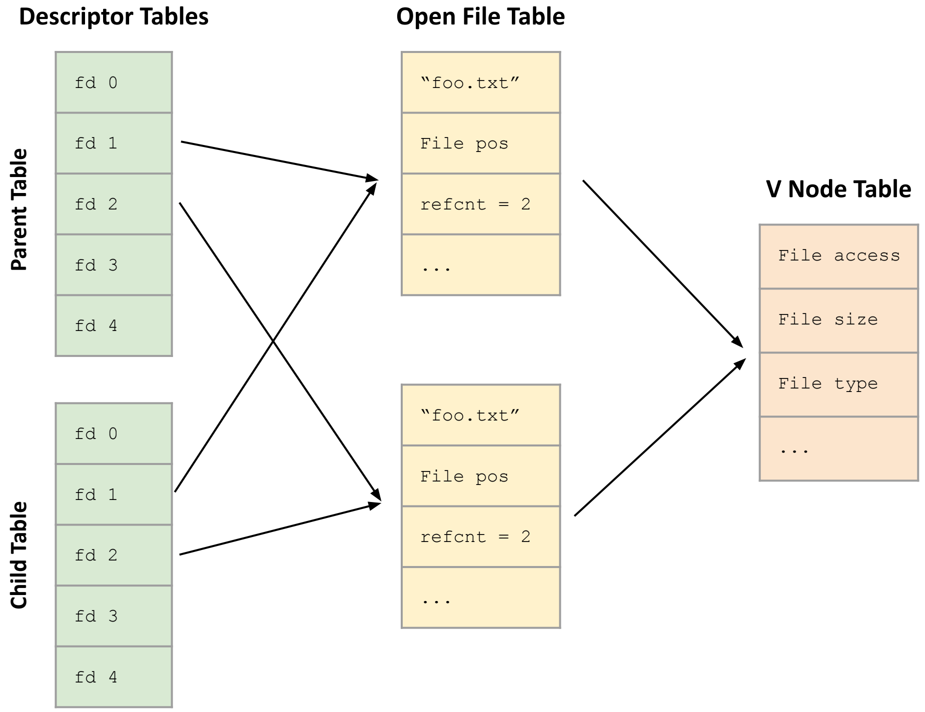
```
void read_and_print_one(int
fd) {
        char c;
        read(fd, &c, 1);
        printf("%c", c);
        fflush(stdout);
}
```

File Descriptors after open() of fd2



```
int main(int argc, char *argv[]) {
        int fd1 = open("foo.txt", O_RDONLY);
   ➡    int fd2 = open("foo.txt", O_RDONLY);
        ...
```

File Descriptors after fork()

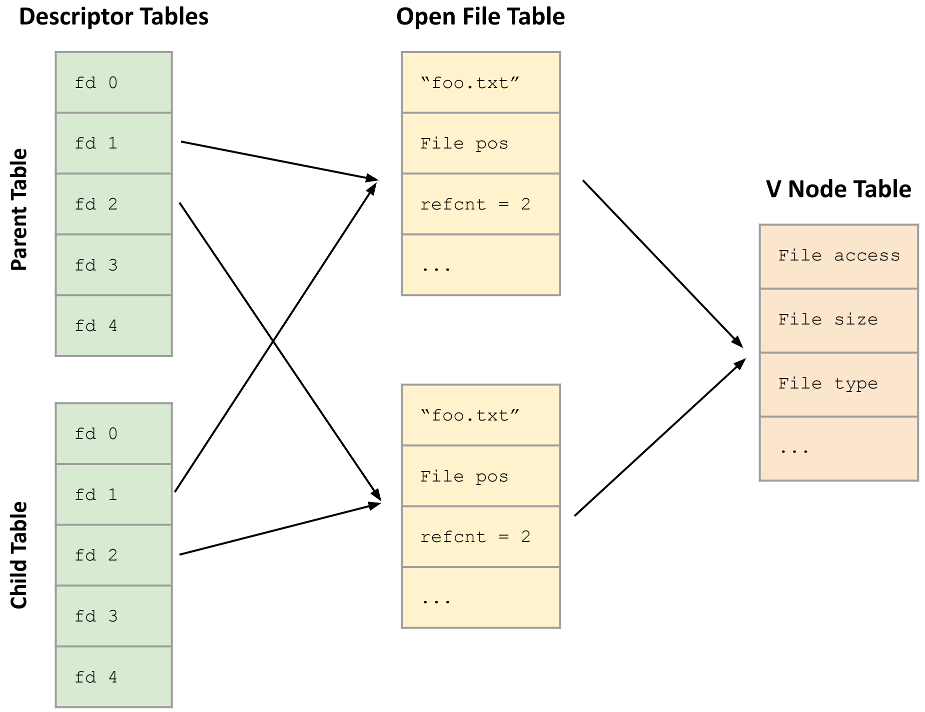


```
int main(int argc, char *argv[]) {
        int fd1 = open("foo.txt", O_RDONLY);
        int fd2 = open("foo.txt", O_RDONLY);
        read_and_print_one(fd1);
        read_and_print_one(fd2);
    →   if(!fork()) {
        ...
```

What has been printed so far?

?

File Descriptors after fork()

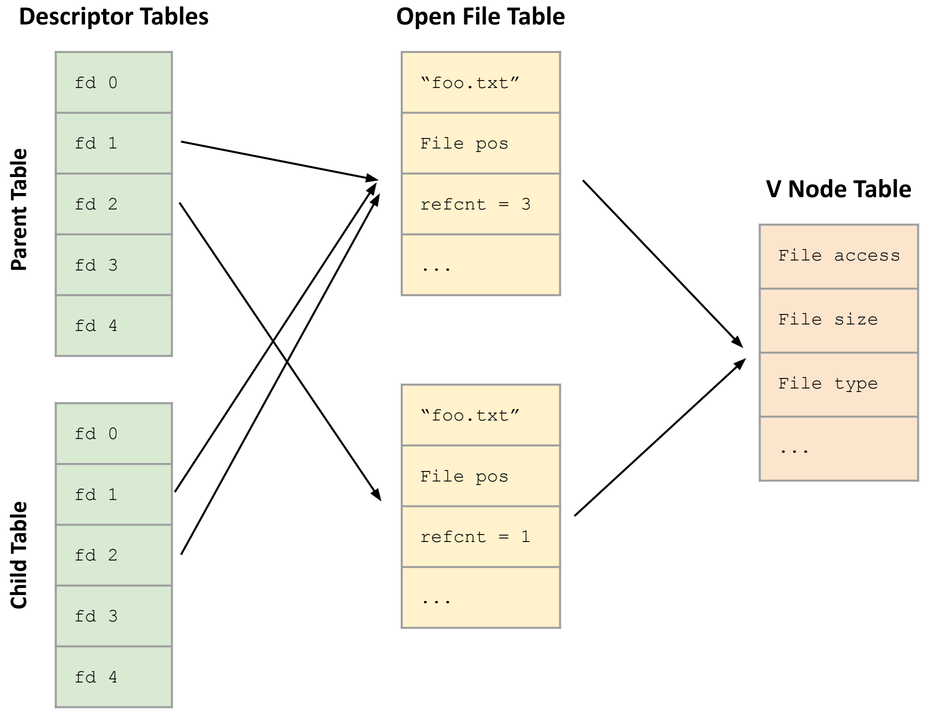


```
int main(int argc, char *argv[]) {
        int fd1 = open("foo.txt", O_RDONLY);
        int fd2 = open("foo.txt", O_RDONLY);
        read_and_print_one(fd1);
        read_and_print_one(fd2);
  →     if(!fork()) {
        ...
```

What has been printed so far?

AA

Output after child prints

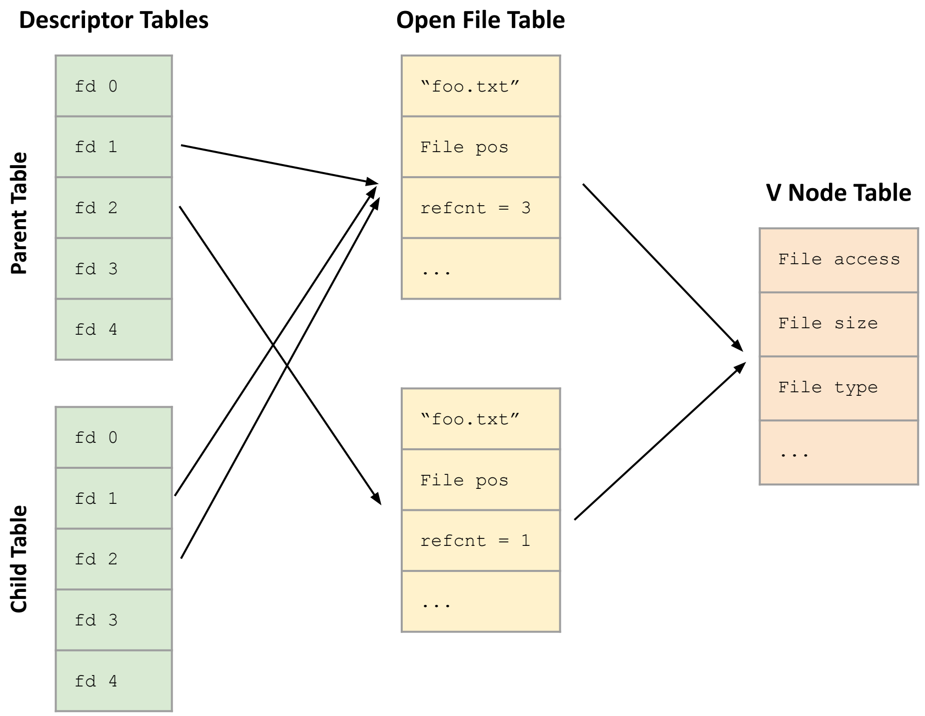


```
int main(int argc, char *argv[]) {
        int fd1 = open("foo.txt", O_RDONLY);
        int fd2 = open("foo.txt", O_RDONLY);
        read_and_print_one(fd1);
        read_and_print_one(fd2);
        if(!fork()) {
                read_and_print_one(fd2);
                read_and_print_one(fd2);
                close(fd2);
                fd2 = dup(fd1);
        →       read_and_print_one(fd2);
        } else {
```

What has been printed so far?

?

Output after child prints

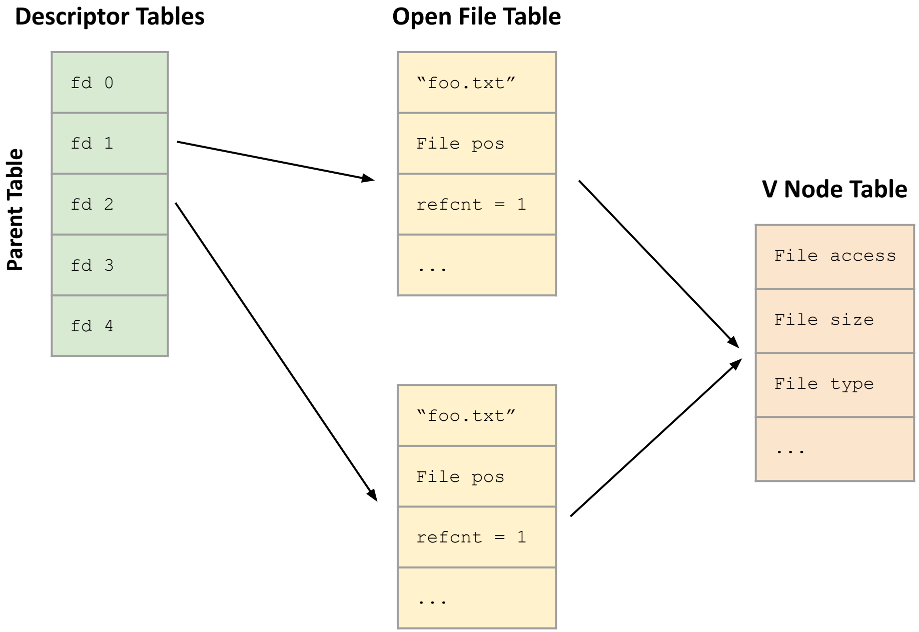


```
int main(int argc, char *argv[]) {
    int fd1 = open("foo.txt", O_RDONLY);
    int fd2 = open("foo.txt", O_RDONLY);
    read_and_print_one(fd1);
    read_and_print_one(fd2);
    if(!fork()) {
        read_and_print_one(fd2);
        read_and_print_one(fd2);
        close(fd2);
        fd2 = dup(fd1);
→       read_and_print_one(fd2);
    } else {
```

What has been printed so far?

AABCB

Output after parent prints



Descriptor Tables

Parent Table

fd 0

fd 1

fd 2

fd 3

fd 4

Open File Table

"foo.txt"

File pos

refcnt = 1

...

"foo.txt"

File pos

refcnt = 1

...

V Node Table

File access

File size

File type

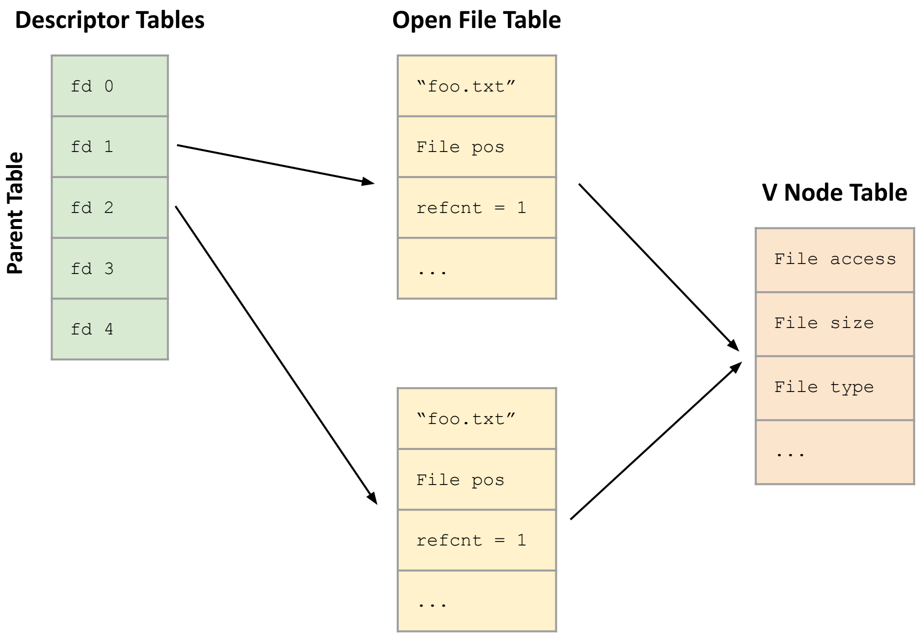
...

```
int main(int argc, char *argv[]) {
    int fd1 = open("foo.txt", O_RDONLY);
    int fd2 = open("foo.txt", O_RDONLY);
    read_and_print_one(fd1);
    read_and_print_one(fd2);
    if(!fork()) {
        read_and_print_one(fd2);
        read_and_print_one(fd2);
        close(fd2);
        fd2 = dup(fd1);
        read_and_print_one(fd2);
    } else {
        wait(NULL);
        read_and_print_one(fd1);
        read_and_print_one(fd2);
    →   printf("\n");
    }
```

What has been printed so far?

?

Output after parent prints



```
int main(int argc, char *argv[]) {
        int fd1 = open("foo.txt", O_RDONLY);
        int fd2 = open("foo.txt", O_RDONLY);
        read_and_print_one(fd1);
        read_and_print_one(fd2);
        if(!fork()) {
                read_and_print_one(fd2);
                read_and_print_one(fd2);
                close(fd2);
                fd2 = dup(fd1);
                read_and_print_one(fd2);
        } else {
                wait(NULL);
                read_and_print_one(fd1);
                read_and_print_one(fd2);
           →    printf("\n");
        }
```

What has been printed so far?

AABCBCD

If you get stuck on tshlab

- **Read the writeup!**
- **Do manual unit testing before** `runtrace` **and** `sdriver`**!**
- **Post private questions on piazza!**

- **Read the man pages on the syscalls.**
  - Especially the error conditions
  - What errors should terminate the shell?
  - What errors should be reported?

man 2 wait

Taken from **http://man7.org/linux/man-pages/man2/wait.2.html**

```
WAIT(2)                    Linux Programmer's Manual                  WAIT(2)

NAME

       wait, waitpid, waitid - wait for process to change state

SYNOPSIS

       #include <sys/types.h>
       #include <sys/wait.h>

       pid_t wait(int *wstatus);

       pid_t waitpid(pid_t pid, int *wstatus, int options);

       int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
                       /* This is the glibc and POSIX interface; see
                          NOTES for information on the raw system call. */
```

man pages (probably) cover all you need

- **What arguments does the function take?**
  - read SYNOPSIS
- **What does the function do?**
  - read DESCRIPTION
- **What does the function return?**
  - read RETURN VALUE
- **What errors can the function fail with?**
  - read ERRORS
- **Is there anything I should watch out for?**
  - read NOTES
- **Different categories for man page entries with the same name**
- **Looking up man pages online is not an academic integrity violation**

Function arguments

- Should I do dup2(old, new) or dup2(new, old)?
- Read the man page:

```
$ man dup2

SYNOPSIS

       #include <unistd.h>

       int dup(int oldfd);
       int dup2(int oldfd, int newfd);
```

Function behavior

- How should I write my format string when I need to print a long double in octals with precision 5 and zero-padded?
- Read the man page

```
$ man printf
```

```
DESCRIPTION
Flag characters
       The character % is followed by zero or more of the following flags:

       #      The value should be converted...
       0      The value should be zero padded...
       -      The converted value is to be left adjusted...
       ' '    (a space) A blank should be left before...
       +      A sign (+ or -) should always ...
```

Function return

- **What does waitpid() return with and without WNOHANG?**
- **Read the man page:**

```
$ man waitpid
```

```
RETURN VALUE

       waitpid(): on success, returns the process ID of the child whose
       state has changed; if WNOHANG was specified and one or more
       child(ren) specified by pid exist, but have not yet changed state,
       then 0 is returned.  On error, -1 is returned.

       Each of these calls sets errno to an appropriate value in the case of
       an error.
```

Potential errors

- **How should I check waitpid for errors?**
- **Read the man page:**

```
$ man waitpid
```

```
ERRORS

       ECHILD (for waitpid() or waitid()) The process specified by pid
              (waitpid()) or idtype and id (waitid()) does not exist or is
              not a child of the calling process.  (This can happen for
              one's own child if the action for SIGCHLD is set to SIG_IGN.
              See also the Linux Notes section about threads.)

       EINTR  WNOHANG was not set and an unblocked signal or a SIGCHLD was
              caught; see signal(7).

       EINVAL The options argument was invalid.
```

Get advice from the developers

- I sprintf from a string into itself, is this okay?
- Read the man page:

```
$ man sprintf
```

```
NOTES
       Some programs imprudently rely on code such as the following

           sprintf(buf, "%s some further text", buf);

       to append text to buf.  However, the standards explicitly note that
       the results are undefined if source and destination buffers overlap
       when calling sprintf(), snprintf(), vsprintf(), and vsnprintf().
       Depending on the version of gcc(1) used, and the compiler options
       employed, calls such as the above will not produce the expected
       results.

       The glibc implementation of the functions snprintf() and vsnprintf()
       conforms to the C99 standard, that is, behaves as described above,
       since glibc version 2.1.  Until glibc 2.0.6, they would return -1
       when the output was truncated.
```