15-213 Recitation
Shell Lab

Your TAs

Friday, November 7th

Reminders

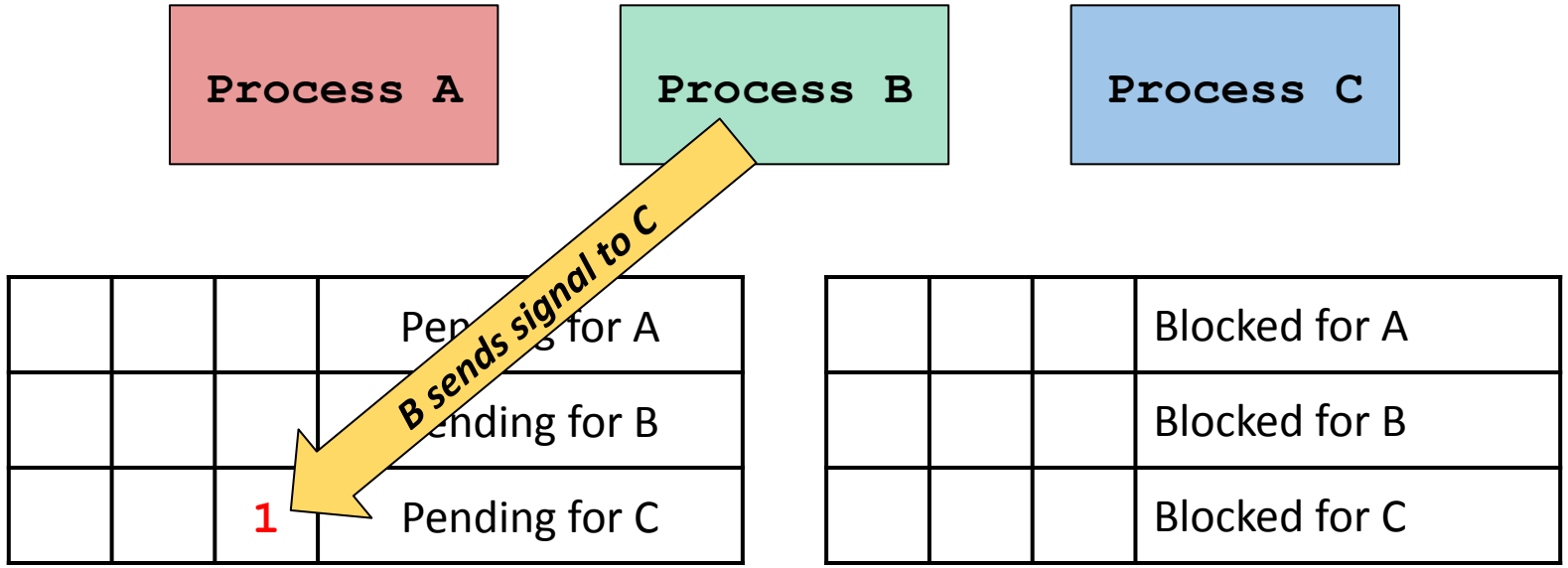
- **`tshlab`** released, due on ***November 18th***.
- ***Written 8*** due ***November 13th***.
- Code Reviews:
  - Watch your inbox for a **`malloc`** Final code review email!

Agenda

- Signals
- File I/O

Signals

Recall: Sending and Receiving Signals



- Pending signals represented by a single *bit*, one for each kind of signal.
- Kernel computes **`pnb`** (pending and not blocked) to determine which signals can be delivered.
- If we don't block them, signals can interrupt our program at any time!

Example: "Counting" with Signals

```
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) { // Do nothing }

    printf("Terminated :-)\n");

}
```

- What happens when we run this program? Will it terminate?
  - **It might not, since signals can *coalesce*.**

Example: "Counting" with Signals

```
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) { // Do nothing }

    printf("Terminated :-)\n");

}
```

Problem: Signals Coalesce

Problem: Tight loop used to wait for signals

- What are some problems with the above code?
- How can we fix them?

Problem: Signals Coalesce

- # Times Signal Handler Called **!=** # Times Signal was Sent
- What can we do instead?

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

Problem: Signals Coalesce

```
void handler(int sig) {
    pid_t pid;

    while ((pid = waitpid(-1, NULL, WNOHANG)) > 0) {
        counter++;
    }
}
```

- **`counter++`** is not atomic. Why aren't there race conditions?

Problem: Efficiently waiting for Signals

- Tight loop is inefficient, and is forbidden in Shell Lab.

```
while (counter < 10) { // Do nothing }
```

Problem: Tight Loop

- Use **`sigsuspend`** instead!

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

- Temporarily installs **`mask`**, then **`pause`**s until a signal is received.
- *Atomic* version of:

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

Recall: Blocking Signals

Textbook: p764

- Allows us to control when our program receives signals.

```
int sigprocmask(int how, sigset_t *mask, sigset_t *prev_mask);
```

- Recommended approach:

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

- Don't use **`SIG_UNBLOCK`**
  - Don't want to unblock a signal if it was already blocked.
  - Allows procedure to be nested multiple times.

Recall: Signal Handlers

- Parent process sends **`SIGINT`** to child process. What is the behavior of the child?
  - No handler specified: *default action.*
  - Can catch the signal with a ***signal handler***.

```
void sigchld_handler(int sig)
{
    int olderrno = errno;
    sigset_t mask_all, prev_all;
    pid_t pid;

    sigfillset(&mask_all);
    while ((pid = waitpid(-1, NULL, 0)) > 0) { /* Reap */
        sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
        deletejob(pid); /* Delete from job list */
        sigprocmask(SIG_SETMASK, &prev_all, NULL);
    }

    if (pid != 0 && errno != ECHILD)
        sio_eprintf("waitpid error");
    errno = olderrno;
}
```

Save and restore `errno`

Temporarily block signals to protect shared data

Call only async-signal-safe functions

Example

```
int main(void) {
    char *tgt = "child";
    sigset_t mask, old_mask;
    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);
    sigprocmask(SIG_BLOCK, &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);
}
```

- How many different lines could be printed?
  - ***0 or 1 line. Parent and child try to terminate each other.***

File I/O

I/O Functions: `open` and `close`

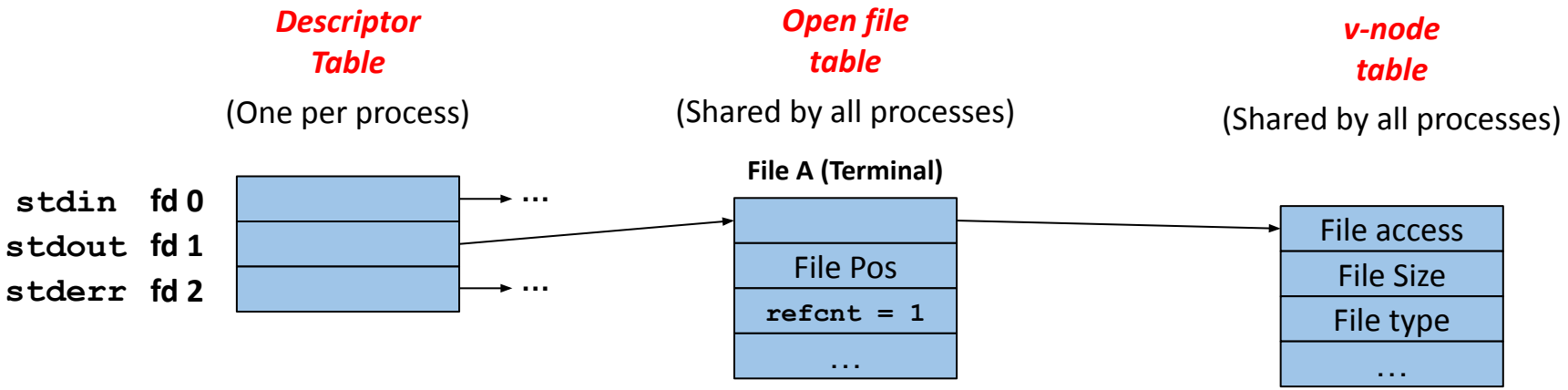
Textbook: p893

```
int open(const char *pathname, int flags, mode_t mode);
```

- **`pathname`** – path to file
- **`flags`**:
  - File Creation: **`O_CREAT`**, **`O_TRUNC`**, etc.
  - Access modes: **`O_RDONLY`**, **`O_WRONLY`**, **`O_RDWR`**
- **`mode`**
  - Specifies who else can read/write the new file.
  - Unless you have reasons not to, use **`DEF_MODE`** from textbook.

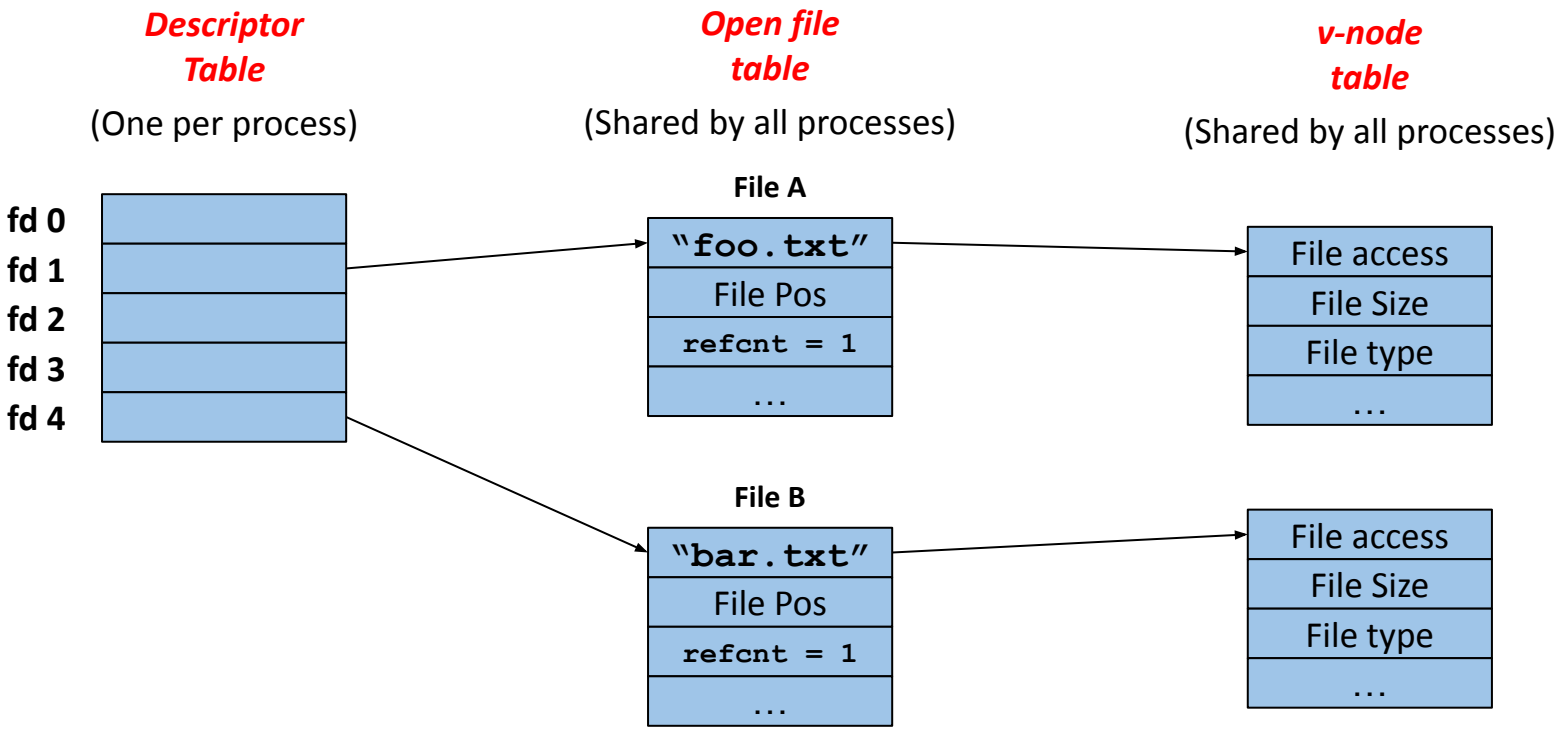
```
int close(int fd);
```

Std File Descriptors

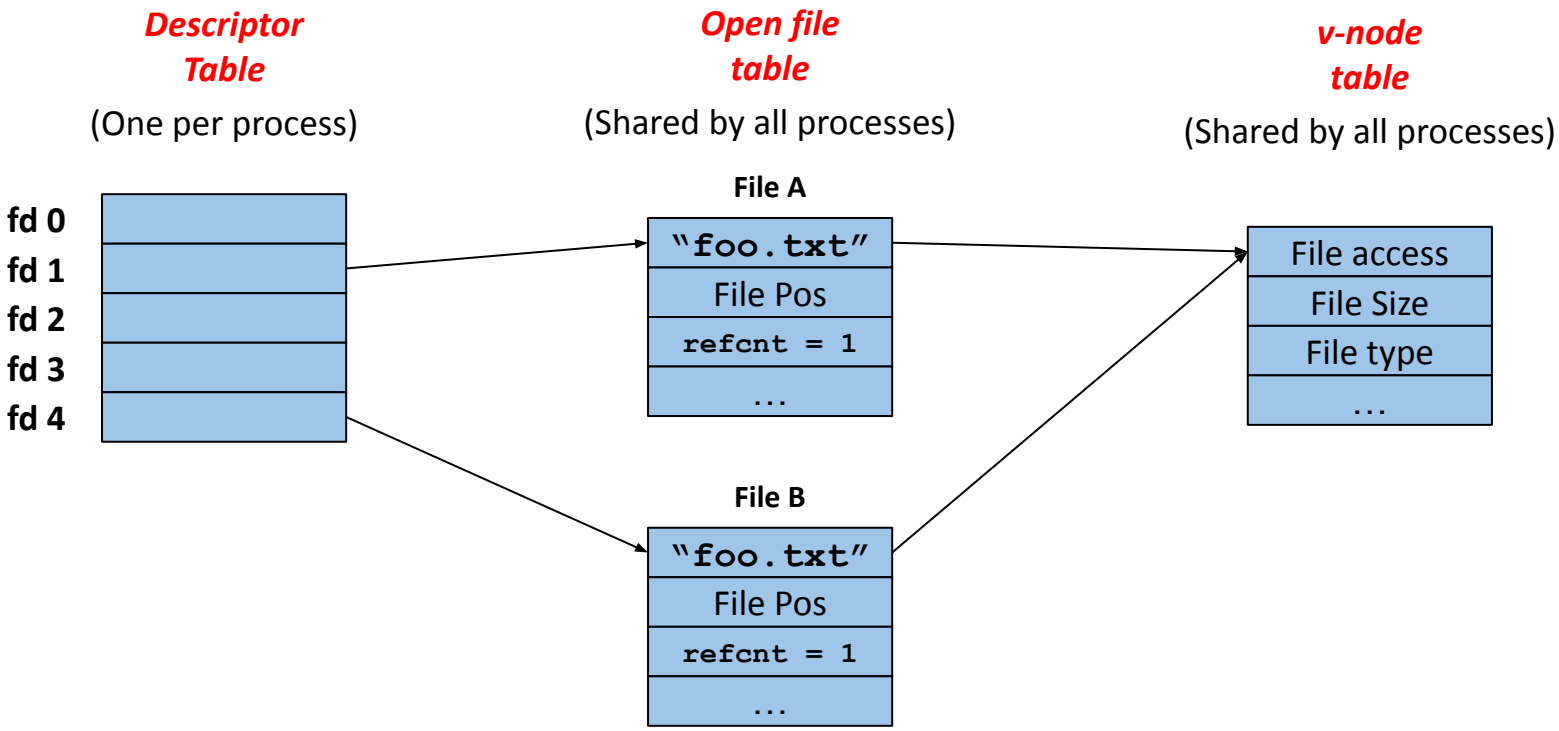


- **`stdin`**, **`stdout`**, and **`stderr`** are set up automatically.
- Closed by normal termination or **`exit()`**.

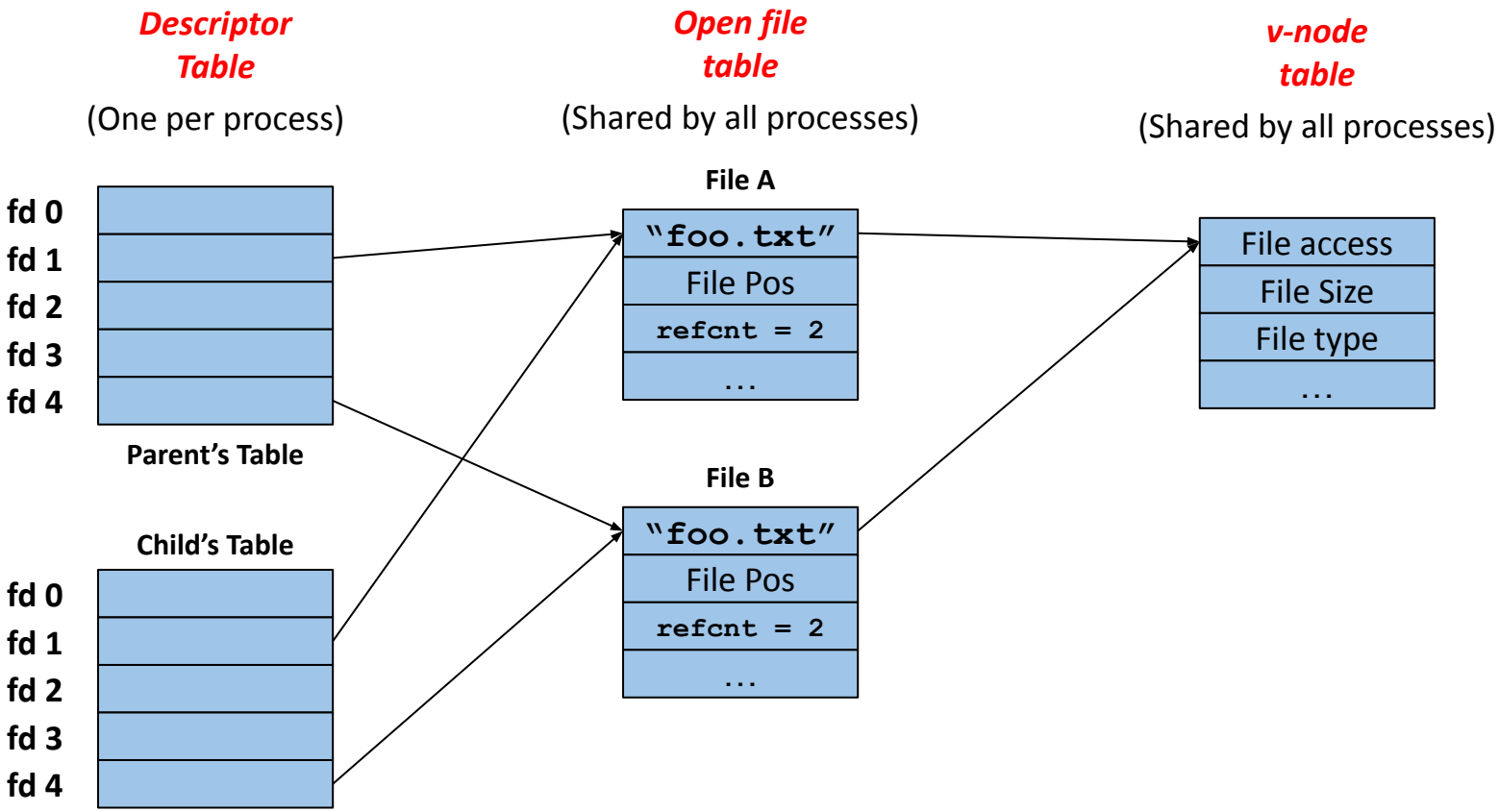
File Descriptors (File A != File B)



File Descriptors (File A == File B)



File Descriptors: What Happens After `fork()`?



Example: I/O and `fork()`

```
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?
```

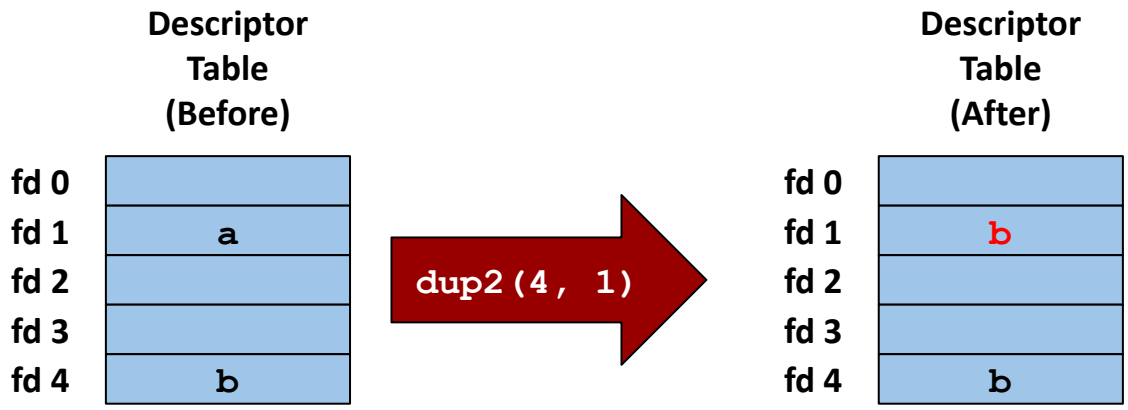
- How many file descriptors are open in the parent process at the indicated point?
- How many does each child have open at the call to **`execve()`**?

I/O Redirection: dup2()

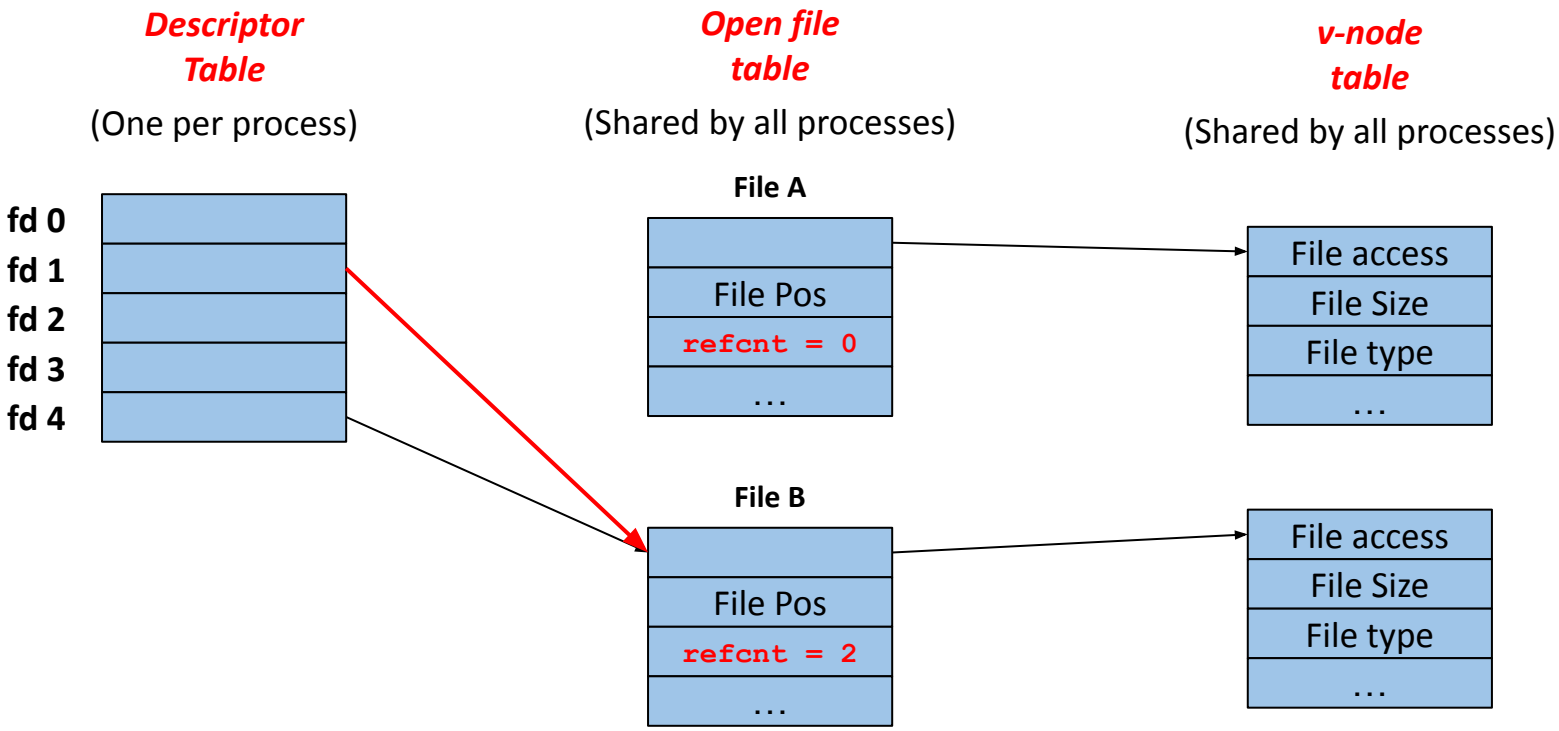
Textbook: p909

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

- Copies descriptor table entry **oldfd** to descriptor table entry **newfd**, overwriting existing entry for **newfd**.



File Descriptors after dup2(4,1)



Initial situation shown on "File A != File B" slide.

Example: Redirecting I/O

```
int main(int argc, char** argv)
{
    int i, fd;
    fd = open("foo", O_WRONLY);

    dup2(fd, STDOUT_FILENO);
    // Point A

    close(fd);
    // Point B
    ...
```

- How many open *file table entries* are there at **Point A**?
- How many open *file table entries* are there at **Point B**?

Example: Redirecting I/O + `fork()`

```
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?
```

- How many file descriptors are open in the parent?

Activity: File I/O

Activity: File I/O

```
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;
}
```

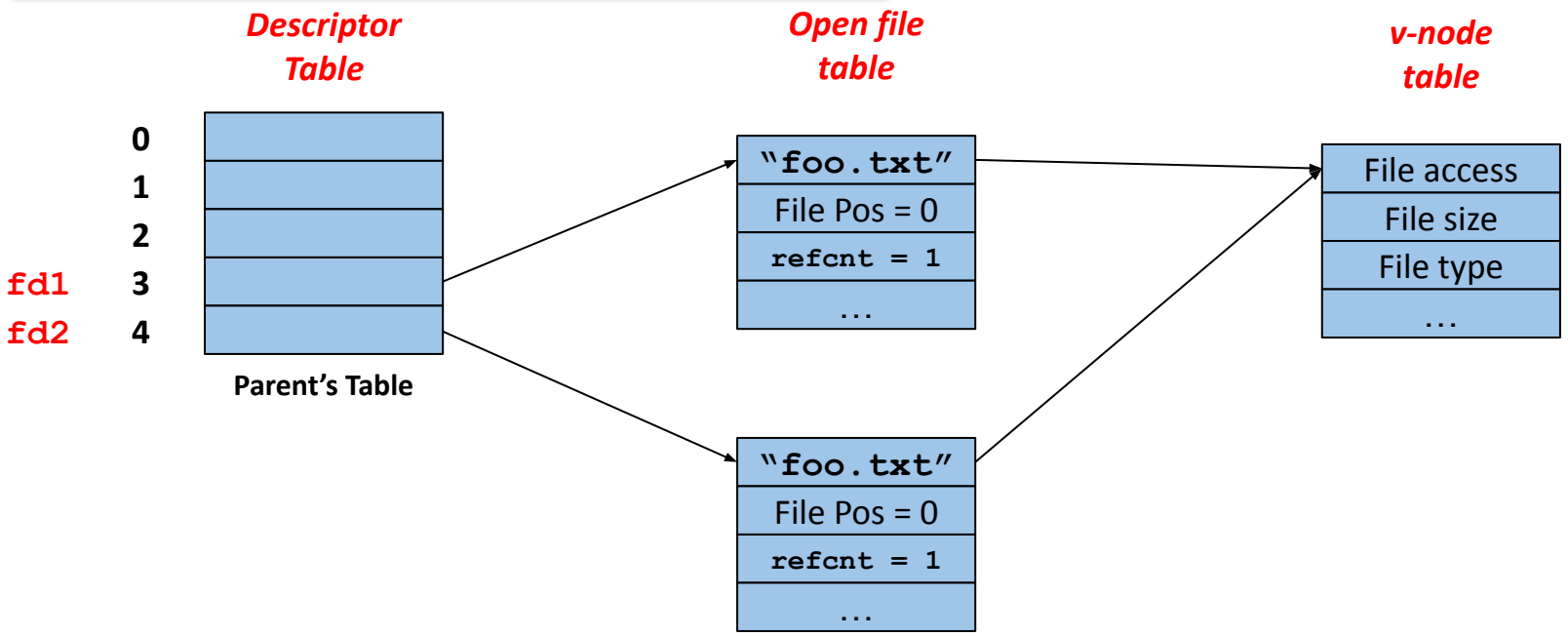
```
void read_and_print_one(int fd) {
     char c;
     read(fd, &c, 1);
     printf("%c", c);
     fflush(stdout);
}
```

- Suppose the contents of foo.txt are "**ABCDEFG**".
- What is the output of this program?

Activity: File I/O

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

- What does the diagram look like at this point?

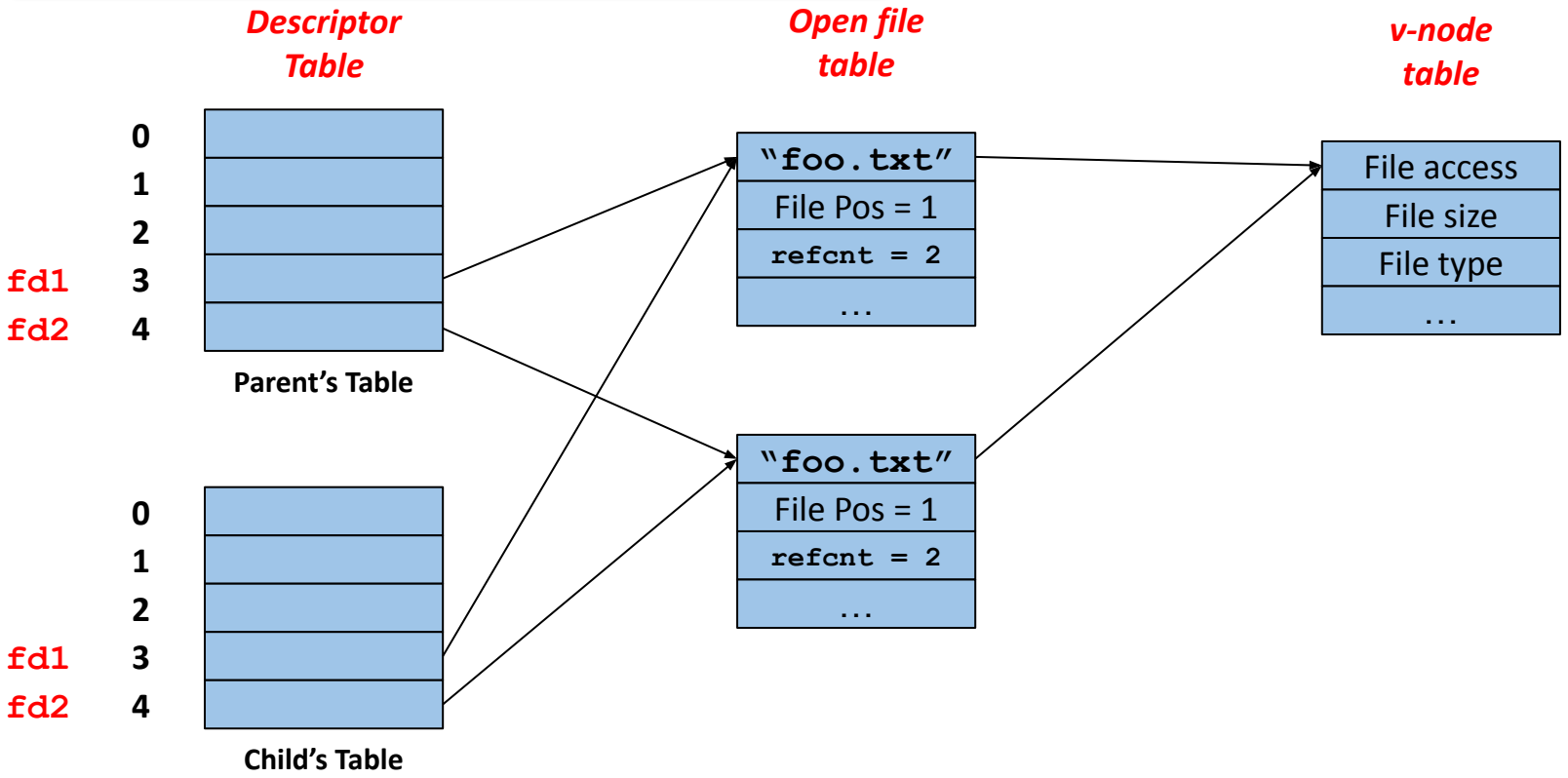


Activity: File I/O

```
...
read_and_print_one(fd1);
read_and_print_one(fd2);

if(!fork()) {
```

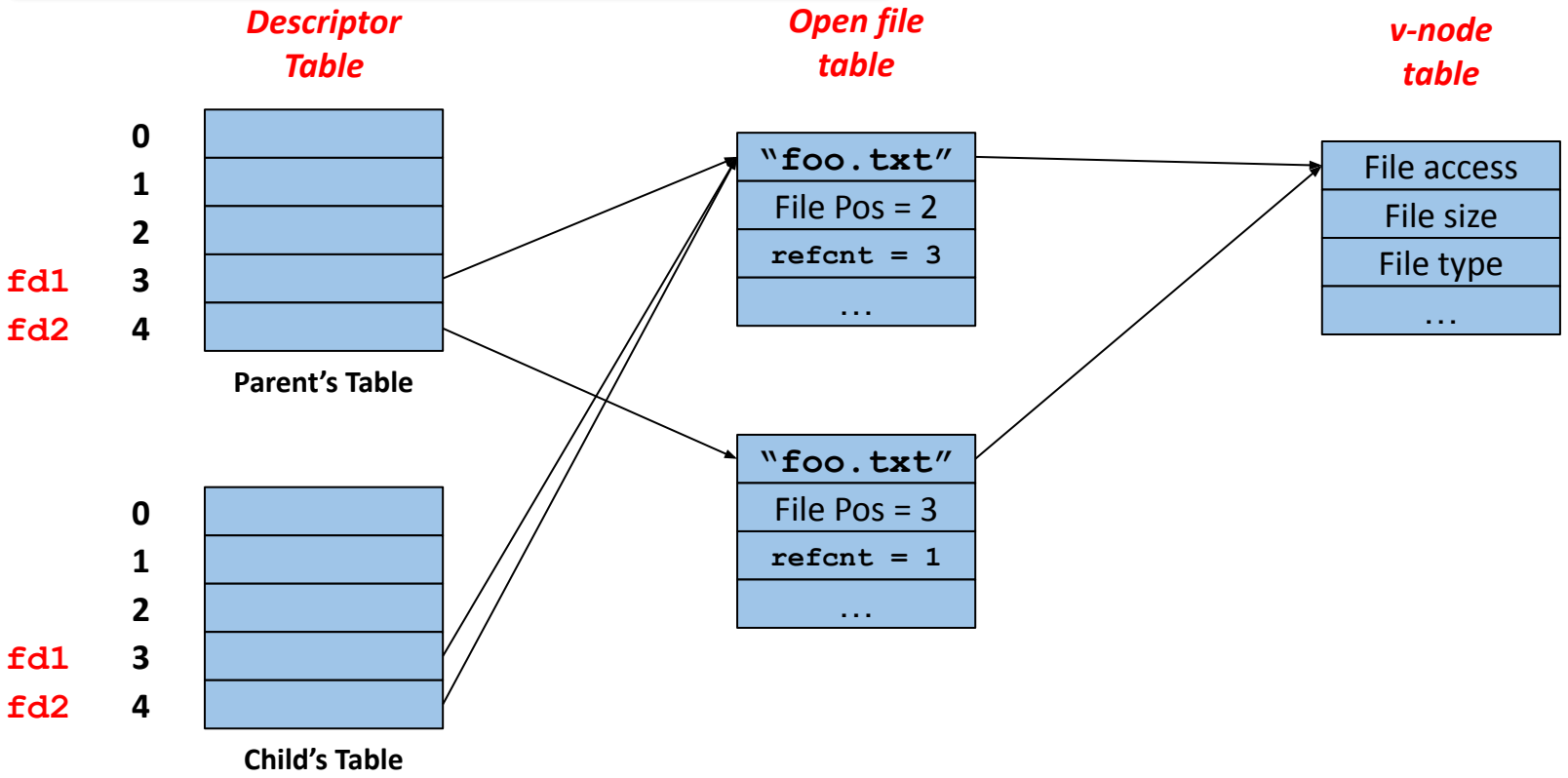
- What has been printed so far?
- **AA**



Activity: File I/O

```
if(!fork()) {
    read_and_print_one(fd2);
    read_and_print_one(fd2);
    close(fd2);
    fd2 = dup(fd1);
    read_and_print_one(fd2);
```

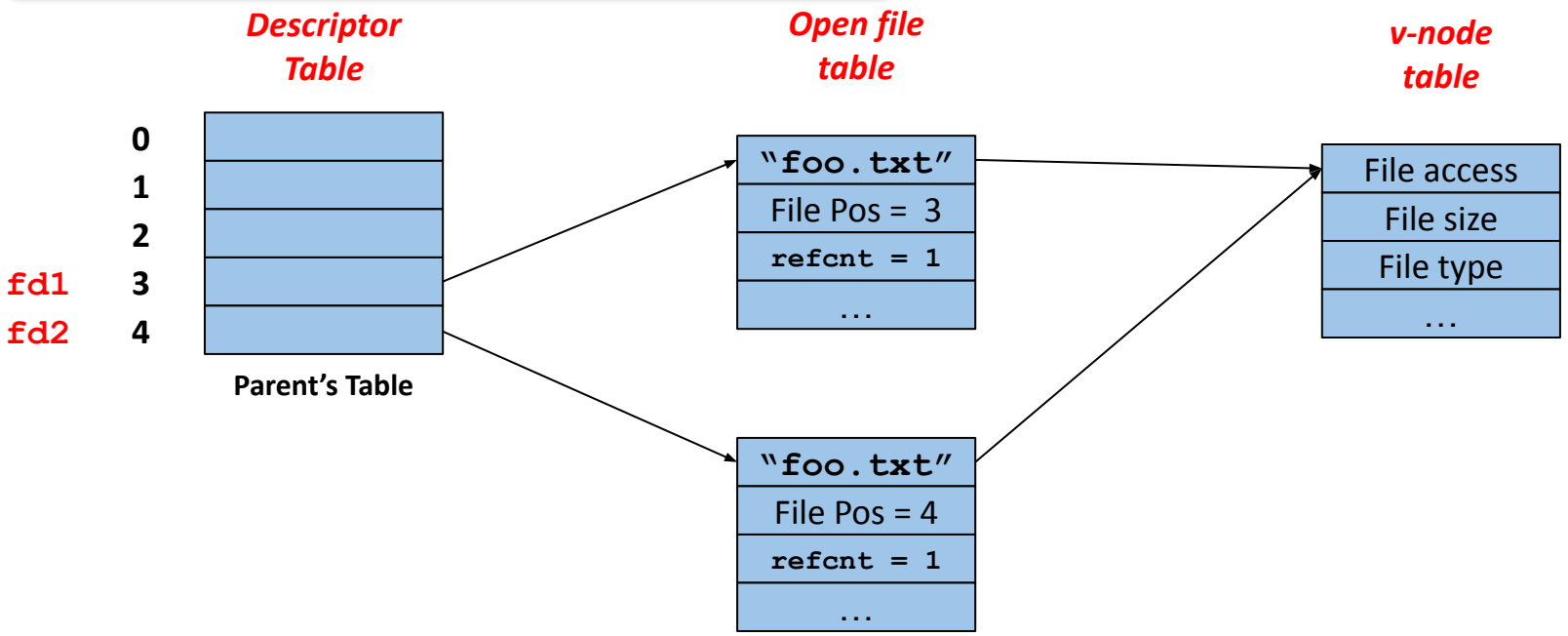
- What has been printed so far?
- **AABCB**



Activity: File I/O

```
    else {
        wait(NULL);
        read_and_print_one(fd1);
        read_and_print_one(fd2);
```

- What has been printed so far?
- **AABCBCD**



Wrapping Up

- **`tshlab`** is due on ***November 18th***
- ***Written 8*** is due on ***November 13th***.
- Code Reviews:
  - Watch your inbox for a **`malloc`** Final code review email!
- Getting Started:
  - Lecture Slides
  - Textbook (Chapter 8)
  - man pages
  - Develop *Incrementally*: see roadmap in write-up.

The End

Shell Lab Reference Sheet

Page numbers refer to CS:APP3e

Loading Programs

- `execve()` [p750]

Waiting and Reaping

- `waitpid()` [p743]
- `sigsuspend()` [p781]
- Exit status macros

  (WIFEXITED, etc.) [p745]

Blocking/Unblocking Signals

- `sigprocmask()` [p764]

Sending Signals

- `kill()` [p761]

I/O

- `open()`, `close()` [p893]
- `dup2()` [p909]