Pipe & File Descriptors

Operating Systems, Fall 2026

Instructors: Travis McGaha

Head TAs: Shriya Sane

Arnuv Batra
Jeffrey Oduman
Kevin Xue
Leo Zhang
Khoi Dinh
Abdullah Goher
Shreya Mukunthan
Sydney Fitzgerald
Monisha Krothapalli
Daniel Price
Rania Souissi
Veer Kakar
Wesley Liu

Poll Everywhere

pollev.com/tqm

- Any questions/comments/concerns?

Administrivia

- First Assignment (penn-vector)
  - Already posted
  - **Real due date is tomorrow 9/09**
  - Mostly a C refresher

- Pre-Semester Survey
  - Anonymous
  - Short!
  - Already posted
  - **Due Wednesday, 9/09**

Administrivia

- Second Assignment (penn-shell)
  - Milestone 0 due Friday 9/11
  - Milestone 1 due Friday 9/18
  - Full assignment due Wednesday 9/23
  - Basic shell implementation
    - System calls, processes, I/O redirection, pipes, job control, and more!
  - Autograders are posted

- **SIGN UP FOR A PARTNER SOMETIME TONIGHT, THE REST WILL BE RANDOMLY ASSIGNED.**
  - **MAKE SURE YOU ARE IN A CANVAS GROUP WITH YOUR PARTNER OR IT WILL BE ASSUMED YOU NEED TO BE RANDOMLY PAIRED**

Lecture Outline

- **wait(), waitpid(), WNOHANG (From Last Week)**
- Intro to file descriptors
- File Descriptors
- Open File Table
- Redirection & Pipes
- Unix Commands & Controls

Review: `wait()`

```
pid_t wait(int *wstatus);
```

- Calling process waits for ***any*** of its children to exit
  - Also *cleans up* the child process
- Gets the exit status of child process through output parameter **`wstatus`**
- Returns process ID of child who was waited for or `-1` on error
- If you need more nuanced behavior, use **`waitpid()`**

Review: `waitpid()`

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

- `pid` is the pid of the child you would like to *check the status of*
  - When `pid` is set to -1, this is equivalent to waiting for any child
- **`wstatus`** *tells us how the child has changed*
- **`options`**, allow us to dictate when waitpid should return
- Returns process ID of child who triggered the return of **`waitpid`** or `-1` on error.
- **`waitpid(-1, &wstatus, 0)` is equivalent to `wait(&wstatus)`**

Options for `waitpid()`

- `pid_t waitpid(pid_t pid, int *wstatus, int options);`
  - **`options`**, allow us to dictate when waitpid should return
    - **`WUNTRACED`: waitpid returns if child was stopped**
    - **`WCONTINUED`: waitpid returns if child was continued (via SIGCONT)**
  - Without these options, **`waitpid`** only returns when a child terminates
  - **`options`** can be or'd together
    - `waitpid(-1, &status, WUNTRACED | WCONTINUED);`

Non blocking wait w/ `waitpid()`

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

- Can pass in **WNOHANG** for **`options`** to make **`waitpid()`** not block or "hang".
- May return
  - process ID of child who triggered the return of **`waitpid`**
  - `-1` on error
  - ***`0` if there are no updates in children processes (specific to WNOHANG)***

Quick NOHANG Demo

- Program will fork a child
- make it sleep for 10 seconds
- parent will call waitpid with the NOHANG flag
- then exit…abandoning it's child…
- When a child has been abandoned, **it becomes an orphan…**

quick_nohang.c

wait/waitpid() status

- **status** output from wait/waitpid() can be checked via macro!
- **WIFEXITED()** true iff the child exited normally via exit or return from main
- **WIFSIGNALED()** true iff the child was terminated via a signal
- **WIFSTOPPED()** true iff the child stopped via delivery of signal
- **WIFCONTINUED()** true iff the child continued via delivery of signal

Why use `wait()/waitpid`? *CPU Utilization*

- When a process is in a blocked state, it will not be run by the scheduler and thus will not use the CPU

- When analyzing performance, one thing people care about is making maximal use of the CPU. The CPU is what is executing our instructions.
  - Avoiding wasting CPU cycles on things that don't matter
  - Make sure the CPU is running as much instructions (that matter) as possible

Poll Everywhere

pollev.com/tqm

```c
int main() {
  pid_t child_pid = fork();

  if (child_pid < 0) {
    perror("Fork failed\n");
    return EXIT_FAILURE;
  }


  if (child_pid == 0) {
    char* argv[] = {"sleep", "10", nullptr};
    execvp(argv[0], argv);
    return EXIT_FAILURE;
  } else {
    int status;
    pid_t res = waitpid(-1, &status, 0);
    while(res == 0) { /* while no updates*/
      printf("Waiting for child...\n");
      res = waitpid(-1, &status, WNOHANG);
    }
    printf("Done waiting for child!\n");
  }
  return EXIT_SUCCESS;
}
```

- What is the output of this program?
  - *Note: Does it behave as we intend?*

no_hang.c

discuss

```c
int main() {
  pid_t child_pid = fork();

  if (child_pid < 0) {
    perror("Fork failed\n");
    return EXIT_FAILURE;
  }


  if (child_pid == 0) {
    char* argv[] = {"sleep", "10", nullptr};
    execvp(argv[0], argv);
    return EXIT_FAILURE;
  } else {
    int status;
    pid_t res = waitpid(-1, &status, WNOHANG);
    while(res == 0) { /* while no updates*/
      printf("Waiting for child...\n");
      res = waitpid(-1, &status, WNOHANG);
    }
    printf("Done waiting for child!\n");
  }
  return EXIT_SUCCESS;
}
```

- Let's change it to use WNOHANG now.
  - *Is this better?*

A common waitpid bug

```
pid_t child_pid = fork();

if(child_pid == 0){ /* Child Only. */
    sleep(atoi(argv[1]));
    return EXIT_SUCCESS;
}

int res_pid = waitpid(child_pid, &status, WNOHANG);

while(!WIFEXITED(status)){

    if(sleeps < 5) {
        printf("Child has not yet excited. Sleeping for a second; will check after I wake up...\n");
        sleep(1);
        sleeps += 1;

    } else if(!signal_sent){
        printf("It's over for you. You're done.\n");
        kill(child_pid, SIGKILL);
        signal_sent = 1;
    }

    int res_pid = waitpid(child_pid, &status, WNOHANG);
}
printf("I have reaped my child.\nGoodbye.\n");
return EXIT_SUCCESS;
```

Initial State:

```
int sleeps = 0;
int signal_sent = 0;
int status = -1;
```

A common waitpid bug

Discuss

```c
pid_t child_pid = fork();

if(child_pid == 0){ /* Child Only. */
    sleep(atoi(argv[1]));
    return EXIT_SUCCESS;
}

int res_pid = waitpid(child_pid, &status, WNOHANG);

while(!WIFEXITED(status)){

    if(sleeps < 5) {
        printf("Child has not yet excited. Sleeping for a second; will check after I wake up...\n");
        sleep(1);
        sleeps += 1;

    } else if(!signal_sent){
        printf("It's over for you. You're done.\n");
        kill(child_pid, SIGKILL);
        signal_sent = 1;
    }

    int res_pid = waitpid(child_pid, &status, WNOHANG);
}
printf("I have reaped my child.\nGoodbye.\n");
return EXIT_SUCCESS;
```

This doesn't work! It runs forever…
What gives? What might be the issue here?

Initial State:

```c
int sleeps = 0;
int signal_sent = 0;
int status = -1;
```

A common waitpid bug

```c
pid_t child_pid = fork();

if(child_pid == 0){ /* Child Only. */
    sleep(atoi(argv[1]));
    return EXIT_SUCCESS;
}

int res_pid = waitpid(child_pid, &status, WNOHANG);

while(!WIFEXITED(status)){

    if(sleeps < 5) {
        printf("Child has not yet excited. Sleeping for a second; will check after I wake up...\n");
        sleep(1);
        sleeps += 1;

    } else if(!signal_sent){
        printf("It's over for you. You're done.\n");
        kill(child_pid, SIGKILL);
        signal_sent = 1;
    }

    int res_pid = waitpid(child_pid, &status, WNOHANG);
}
printf("I have reaped my child.\nGoodbye.\n");
return EXIT_SUCCESS;
```

What gives? What might be the issue here?

Remember: When is WIFEXITED true?

- WIFEXITED is true only when a process terminates normally!
  - (Via Return from Main, or EXIT()…)
- What should we check in addition to this?

A common waitpid bug

```c
pid_t child_pid = fork();

if(child_pid == 0){ /* Child Only. */
    sleep(atoi(argv[1]));
    return EXIT_SUCCESS;
}

int res_pid = waitpid(child_pid, &status, WNOHANG);

while(!WIFEXITED(status) && !WIFSIGNALED(status)){  /* Remember WIFEXITED only exits if the program terminated normally.*/

    if(sleeps < 5) {
        printf("Child has not yet excited. Sleeping for a second; will check after I wake up...\n");
        sleep(1);
        sleeps += 1;

    } else if(!signal_sent){
        printf("It's over for you. You're done.\n");
        kill(child_pid, SIGKILL);
        signal_sent = 1;
    }

    int res_pid = waitpid(child_pid, &status, WNOHANG);
}
printf("I have reaped my child.\nGoodbye.\n");
return EXIT_SUCCESS;
```

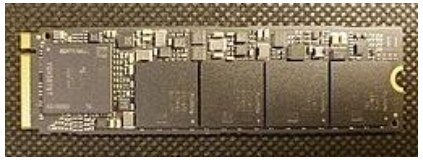
This fixes our check, so we exit when either the program has terminated normally or via our sent signal! Neat.

Lecture Outline

- wait(), waitpid(), WNOHANG (From Last Week)
- **Intro to file descriptors**
- File Descriptors
- Open File Table
- Redirection & Pipes
- Unix Commands & Controls

What is a File?

- Files are "non-volatile storage" that are external to a process:
  - Changes to a file persist beyond the lifetime of a process
  - The same file can be access by multiple processes
  - Stored on completely different hardware than normal process memory



- More details on Files later…

What is a file descriptor?

- A file descriptor is of type `int`
  - A process specific unique id that can be used to refer to a file when invoking system calls
- A file descriptor may not refer to a *literal* file, but instead refer to something that is "like a file"
  - Terminal input/output
  - Network connections
  - Pipes (more later this lecture)
  - Special devices
- These can all be used for **`read()`** and **`write()`**

`stdin`, `stdout`, `stderr`

- By default, there are three "files" open when a program starts
  - `stdin`: for reading terminal input typed by a user
    - `stdin` in C `stdio.h`
    - `System.in` in Java
    - `sys.stdin` in Python
  - `stdout`: the normal terminal output (buffered)
    - `stdout` in C `stdio.h`
    - `System.out` in Java
    - `sys.stdout` in Python
  - `stderr`: the terminal output for printing errors (unbuffered)
    - `stderr` in C `stdio.h`
    - `System.err` in Java
    - `sys.stdin` in Python

Note: `stdin`, `stdout`, and `stderr` are pointers to a `FILE` struct in C.

stdin, stdout, stderr

- stdin, stdout, and stderr all have initial file descriptors constants defined in unistd.h

unistd.h

Library > Developer > CommandLineTools > SDKs > MacOSX.sdk > usr > include > unistd.h > ...

```
68  #ifndef _UNISTD_H_
85
86  #define  STDIN_FILENO   0   /* standard input file descriptor */
87  #define STDOUT_FILENO   1   /* standard output file descriptor */
88  #define STDERR_FILENO   2   /* standard error file descriptor */
89
```

- These will be allocated on default for a process

- Printing to standard out with printf will use write(STDOUT_FILENO, …)

open()

```
int open(const char* pathname, int flags, /* mode_t perm */ )
```

- pathname
  - The name of the file you'd like you 'open'
- flags
  - Bitwise OR specifying behavior for the opening of the file
  - **MUST INCLUDE ONE OF THESE: O_RDONLY, O_WRONLY, O_RDWR**
- perm
  - Used only if flag O_CREAT is used to specify the set of permissions allowed in interacting with file
    - E.g. is the file, executable?, who can see it?, etc.
  - Called mode in the man pages (but that's kind of misleading…)
- Returns:
  - A valid file descriptor or -1 on error

Quick Aside: Permissions

```
int open(const char* pathname, int flags, /* mode_t perm */ )
```

- perm
  - Used only if flag O_CREAT is used
  - Specify the set of permissions allowed in interacting with file
    - E.g. is the file, executable?, who can see it?, etc.
- Permissions are kept in three octets
  - One for the **User**, the **Group**, and **Other**.
  - r - read
  - w - write
  - x - execute

Quick Aside: Permissions

```
int open(const char* pathname, int flags, /* mode_t perm */ )
```

- perm
  - Used only if flag O_CREAT is used
  - Specify the set of permissions allowed in interacting with file
    - E.g. is the file, executable?, who can see it?, etc.

User	Group	Other	
rwx	r-x	r-x	delay_sigint
1 1 1	1 0 1	1 0 1	
0x7	0x5	0x5	

The intricacies of what *consists of the group and other is irrelevant in this course.*

In assignments, you will use the 0744 (rwxr—r--) octet when creating new files.

close()

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

- fd
  - The corresponding file descriptor that will be closed.
  - No longer will this number refer to the file.
  - Allows it for it to be *reused*
- *Imperative once we talk about pipes and file references…*

Using open() & close()

```c
int main(int argc, char *argv[]){
    // Open the specified source file for reading
    int opened_file_fd = open(argv[1], O_RDONLY);

    // Do some magic with the file descriptor

    close(opened_file_fd);
    return 0;
}
```

For Completeness: other FD system calls.

```
ssize_t read(int fd, void buf[.count], size_t count);
```

- Attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.
- On success, the number of bytes read is returned. On error, -1 is returned, and *errno* is set to indicate the error. 0 is returned when EOF has been encountered (usually)…

```
ssize_t write(int fd, const void buf[.count], size_t count);
```

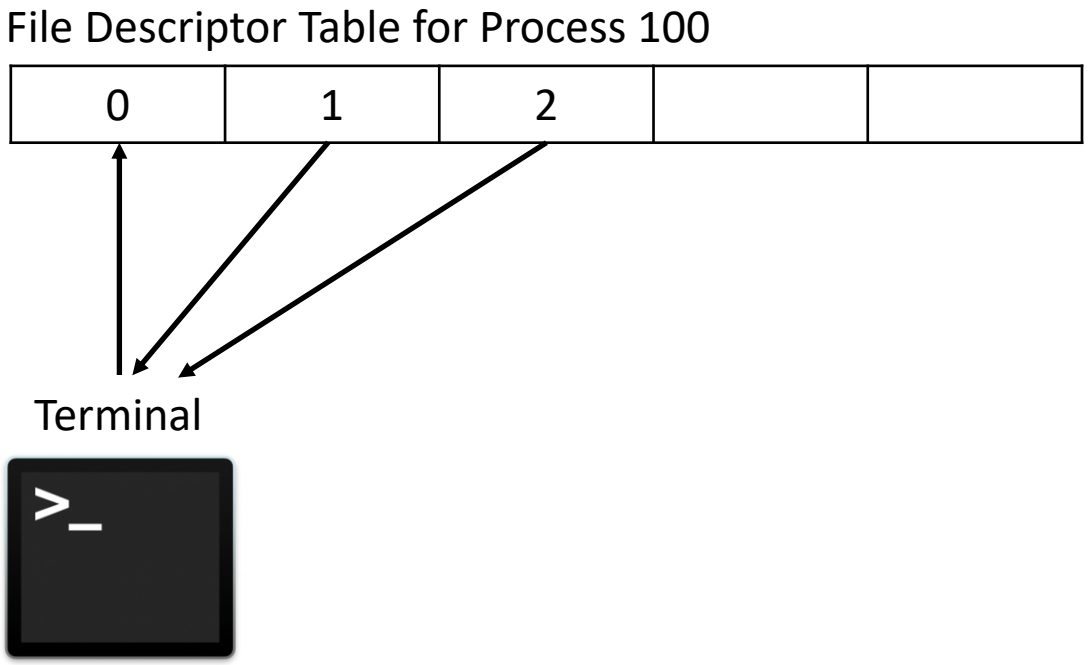
- Writes up to *count* bytes from the buffer starting at *buf* to the file referred to by the file descriptor *fd*.
  - On success, the number of bytes written is returned. On error, -1 is returned, and *errno* is set to indicate the error.

Lecture Outline

- wait(), waitpid(), WNOHANG (From Last Week)
- Intro to file descriptors
- **File Descriptors**
- Open File Table
- Redirection & Pipes
- Unix Commands & Controls

File Descriptor Table

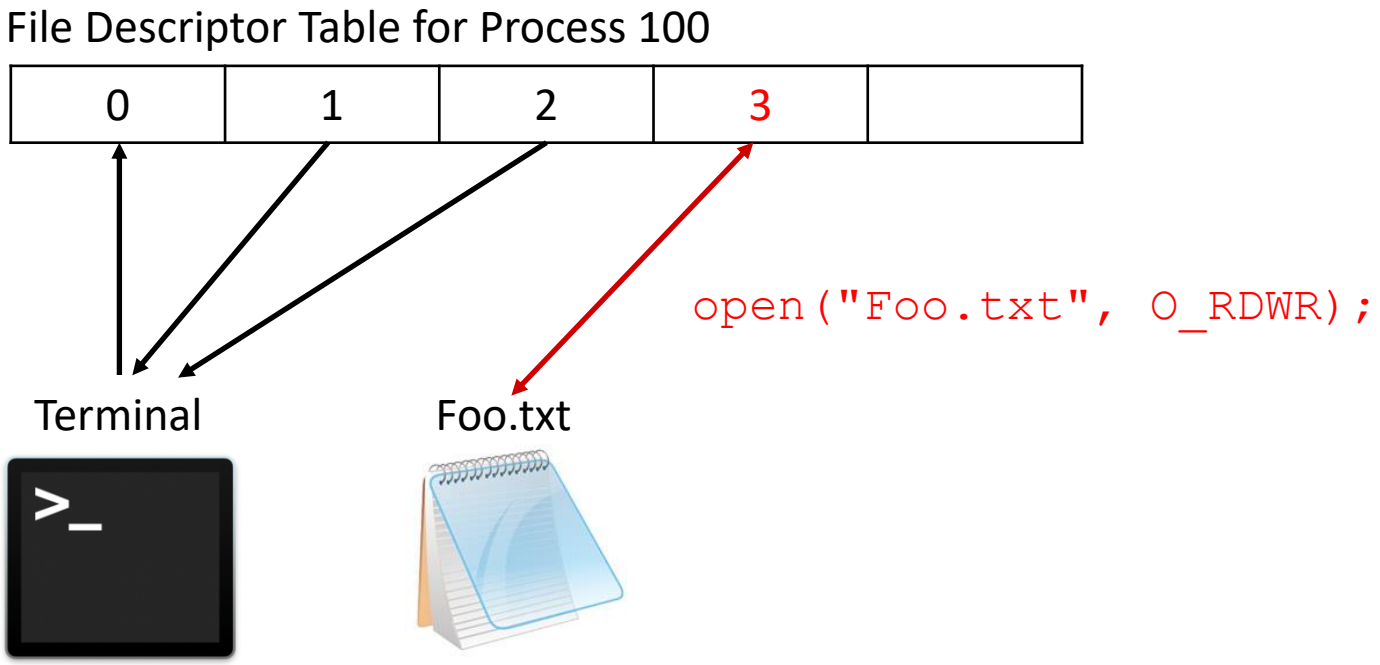
- Each process has **its own file descriptor table** managed by the OS
  - The table maintains information about the respective files the process has references to
- A *file descriptor* is an index into a processes FD table.



Not an accurate depiction.

File Descriptor Table

- Each process has **its own file descriptor table** managed by the OS
  - The table maintains information about the respective files the process has references to
- A *file descriptor* is an index into a processes FD table.



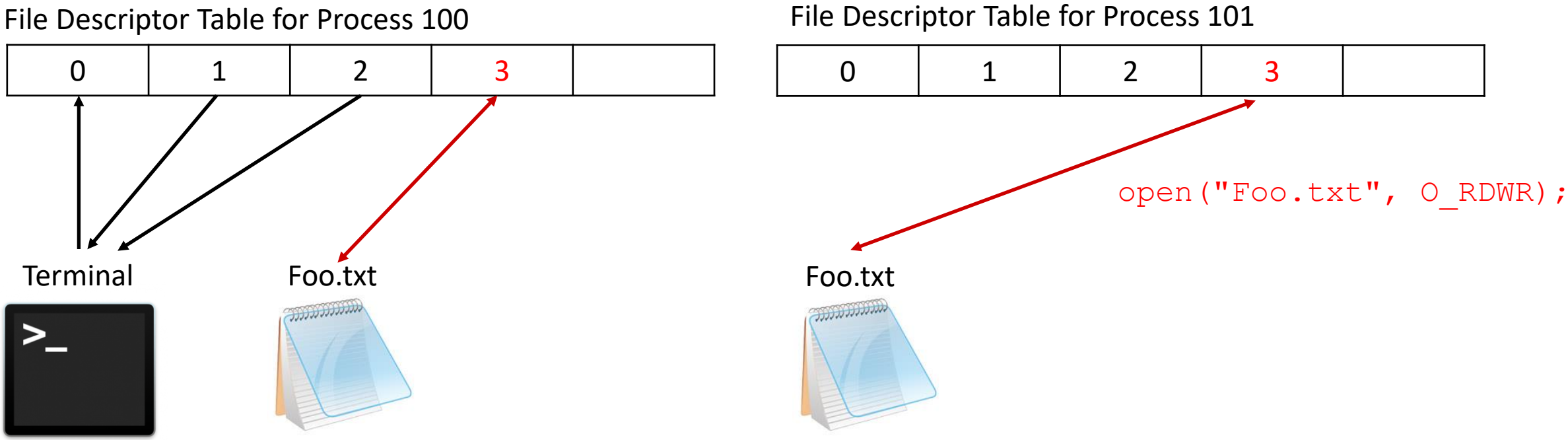
Not an accurate depiction.

Note: the lowest # fd is given when possible.

File Descriptor Table

- Each process has **its own file descriptor table** managed by the OS
  - The table maintains information about the respective files the process has references to
- A *file descriptor* is an index into a processes FD table.

If another process opens the same file, they have a separate file descriptor to the same file



Not an accurate depiction.

Note: the lowest # fd is given when possible.

Poll Everywhere

Discuss

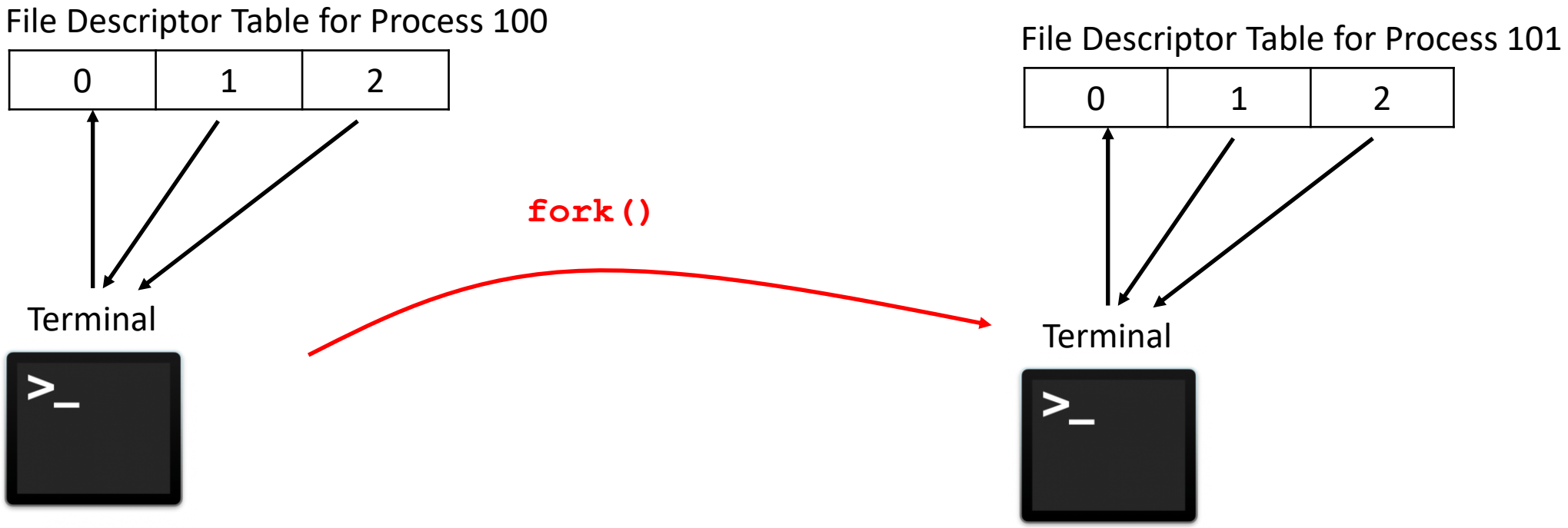
- What if there was only one global file descriptor table? What negative affects may this have?

File Descriptor Table for Process 100
File Descriptor Table for Process 101
File Descriptor Table for Process 102
File Descriptor Table for Process 103
File Descriptor Table for Process 104
File Descriptor Table for Process 105

0	1	2	3	

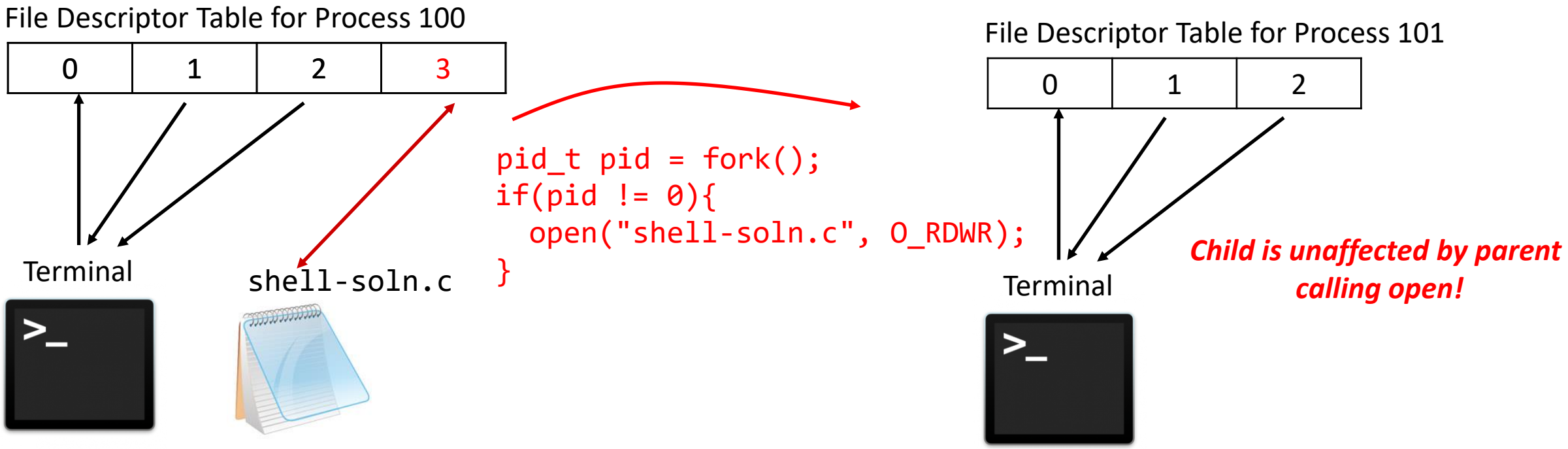
File Descriptor Table and Fork

- Each process will have **its own file descriptor table** managed by the OS
- Fork will make a copy of the parent's file descriptor table for the child



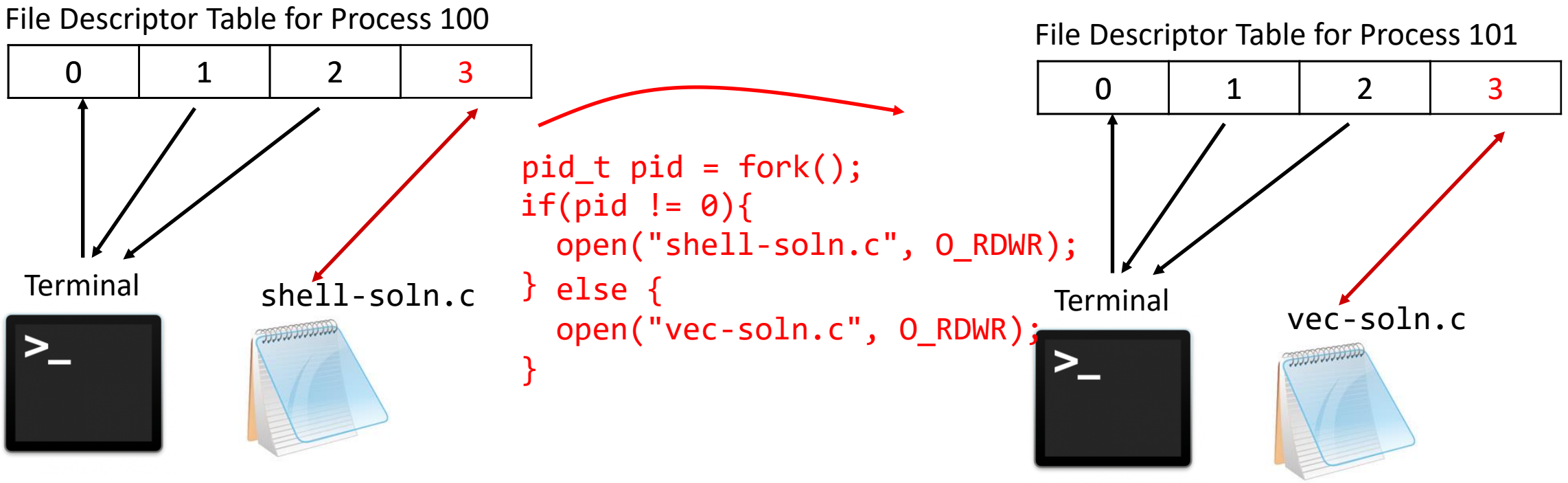
Example: open After fork

- Each process will have **its own file descriptor table** managed by the OS
- Fork will make ***a copy of the parent's file descriptor table for the child***



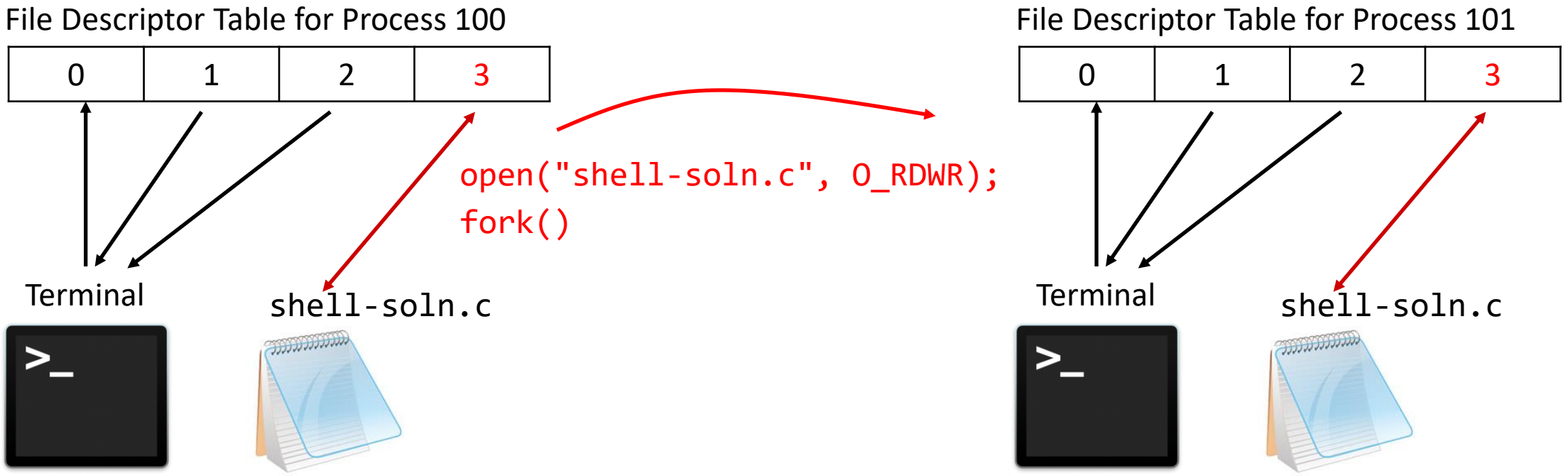
Example: open After fork

- Each process will have **its own file descriptor table** managed by the OS
- Fork will make ***a copy of the parent's file descriptor table for the child***



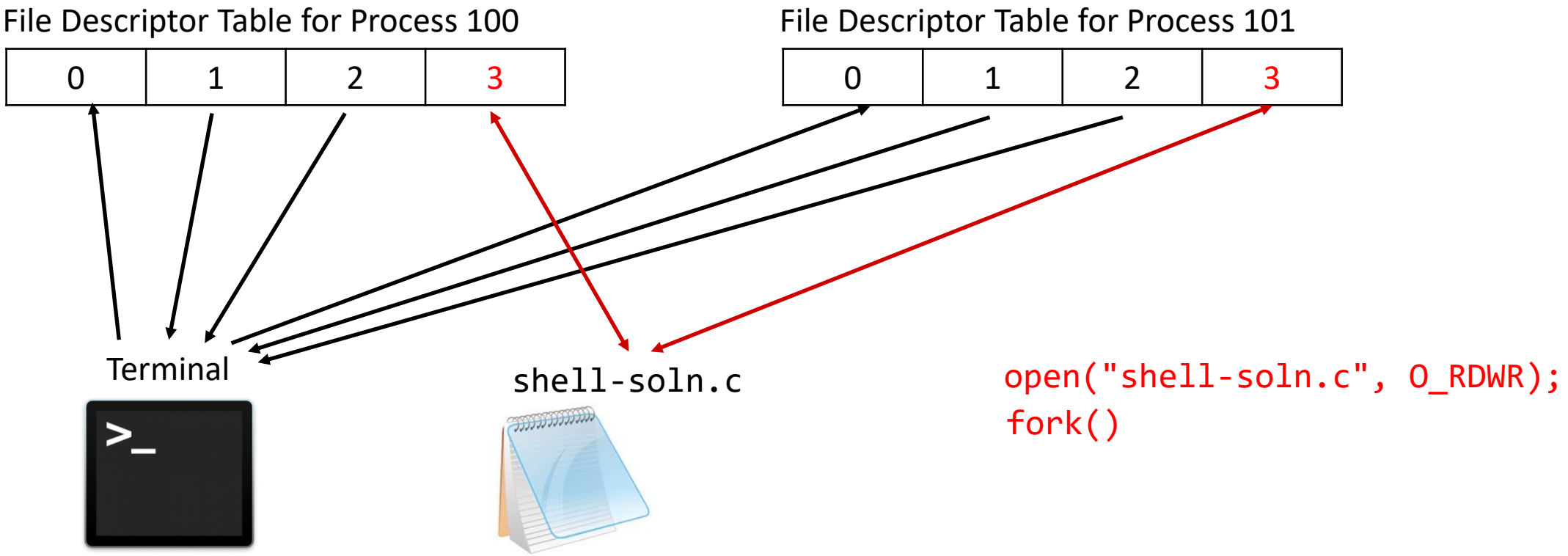
Example: open Before fork

- Fork will make an IDENTICAL copy of the parent's file descriptor table
- ***This seems like overkill – we have the same file opened twice…***



The Inherited Descriptor Table

- Fork will make an IDENTICAL copy of the parent's file descriptor table
- **A somewhat more accurate diagram…. ☺**



File Descriptor Table Summary

- Each process has **its own file descriptor table** managed by the OS
- A *file descriptor* is an index into a processes FD table
- Children made via fork inherent an identical FD table from their parent
  - Those files are not closed nor are they modified in any way. They are the exact same files referred to by the parent.
- Files opened exclusively in a process after a fork do not modify the FD table of another process, child or parent

Lecture Outline

- wait(), waitpid(), WNOHANG (From Last Week)
- Intro to file descriptors
- File Descriptors
- **Open File Table**
- Redirection & Pipes
- Unix Commands & Controls

The Open File Table

- Each process has its own file descriptor table
  - We index into the file descriptor table using the file descriptor

Process 100

File Descriptor Table

0	1	2	3	4	5	…
ptr	ptr	ptr	ptr	ptr	ptr	…

Each entry in the FD Table is a pointer to a system wide file table maintained by the kernel!

As we open more files, using open(), not only do we receive an FD but an entry is (potentially) made in the open file table!

mode	Read	mode	Write	mode	Write	mode	….
cursor (offset position)	0	*cursor (offset position)*	0	*cursor (offset position)*	0	*cursor (offset position)*	….
reference count	1	reference count	1	reference count	1	reference count	…
File Name (path)	file_a.txt	File Name (path)	file_b.txt	File Name (path)	file_c.txt	File Name (path)	…
vnode (FileSys Info)	….	vnode (FileSys Info)	….	vnode (FileSys Info)	….	vnode (FileSys Info)	…

The Open File Table

- Each process has its own file descriptor table
  - We index into the file descriptor table using the file descriptor

Process 100

File Descriptor Table

0	1	2	3	4	5	…
ptr	ptr	ptr	ptr	ptr	ptr	…

cursor: keeps track of *where in the file we are either "reading" or writing" to*

ref_count: the number of references to that entry in the File Table

file_name: the "name" of the corresponding file

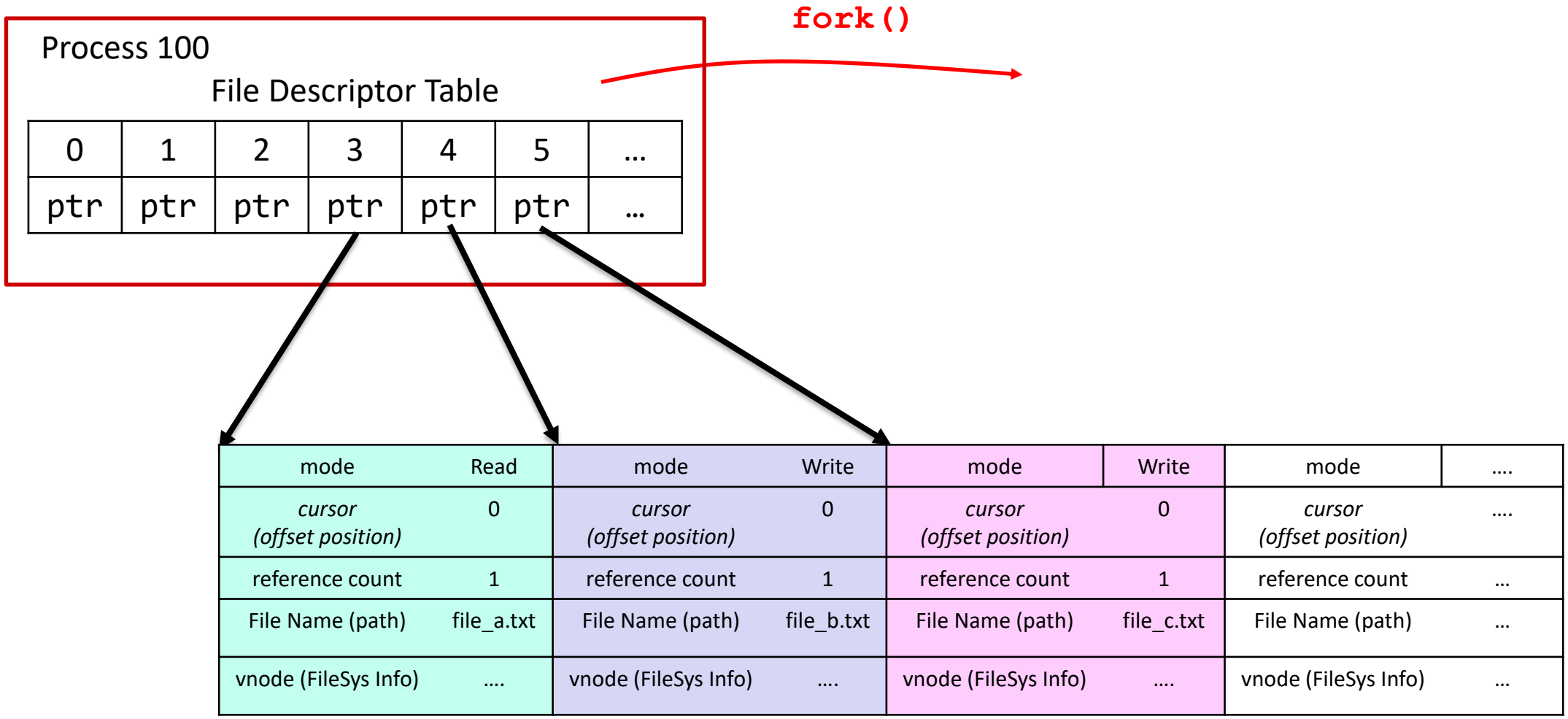
vnode/inode: don't worry yet….

This is what f/lseek() manipulates!

mode	Read	mode	Write	mode	Write	mode	….
cursor (offset position)	0	*cursor (offset position)*	0	*cursor (offset position)*	0	*cursor (offset position)*	….
reference count	1	reference count	1	reference count	1	reference count	…
File Name (path)	file_a.txt	File Name (path)	file_b.txt	File Name (path)	file_c.txt	File Name (path)	…
vnode (FileSys Info)	….	vnode (FileSys Info)	….	vnode (FileSys Info)	….	vnode (FileSys Info)	…

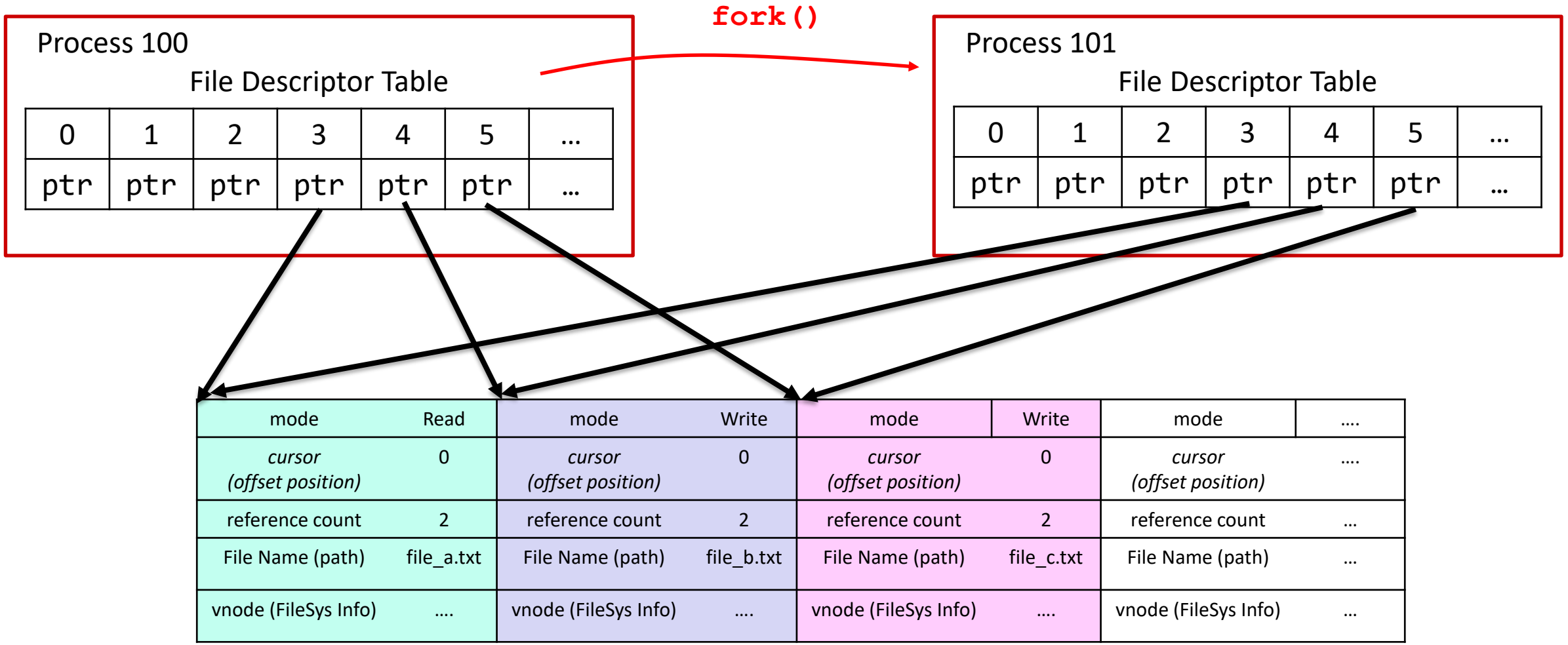
File Descriptor Table and Fork

- When a process forks, the child inherits an identical FD Table!

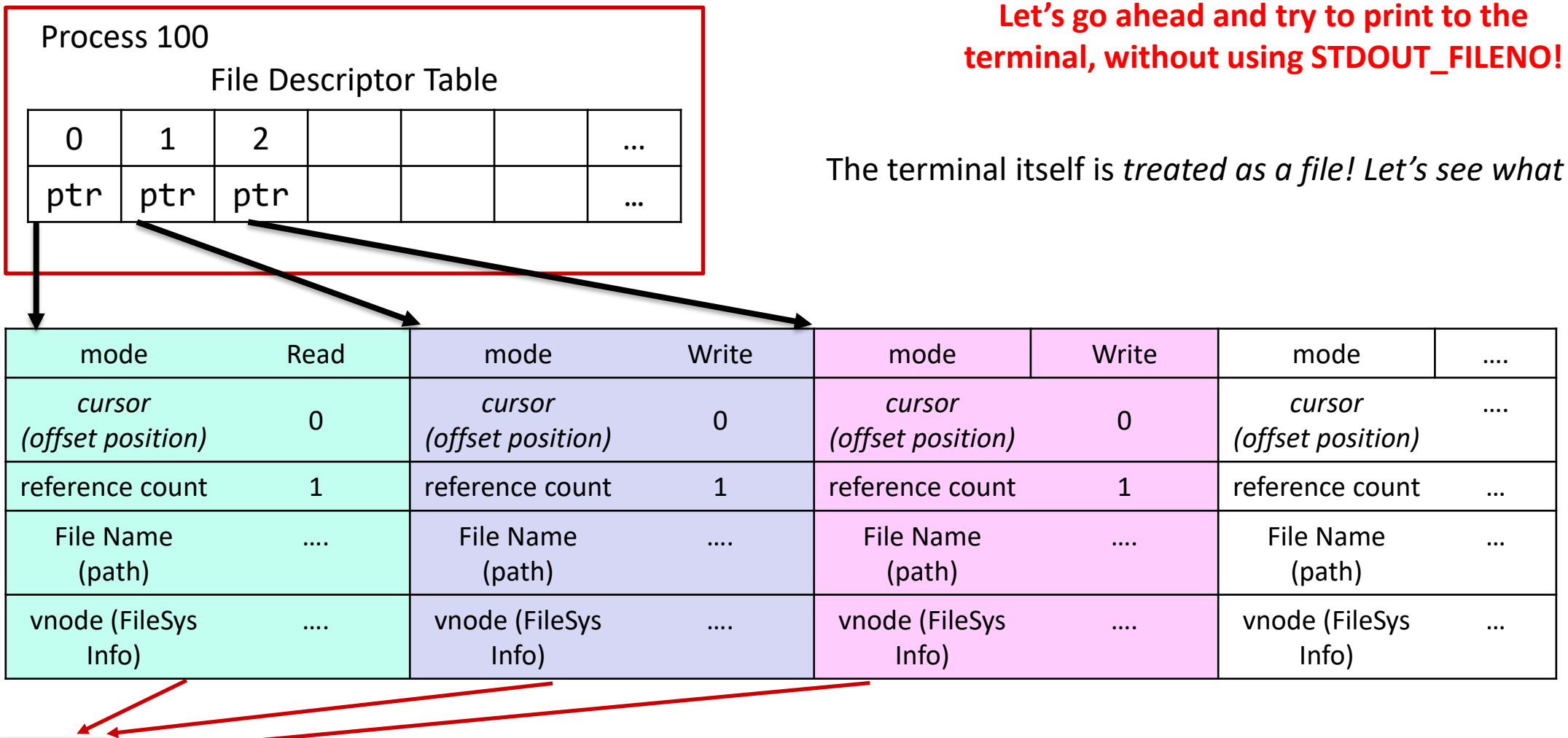


File Descriptor Table and Fork

- When a process forks, the child inherits an identical FD Table!



Demo: Terminal Printing

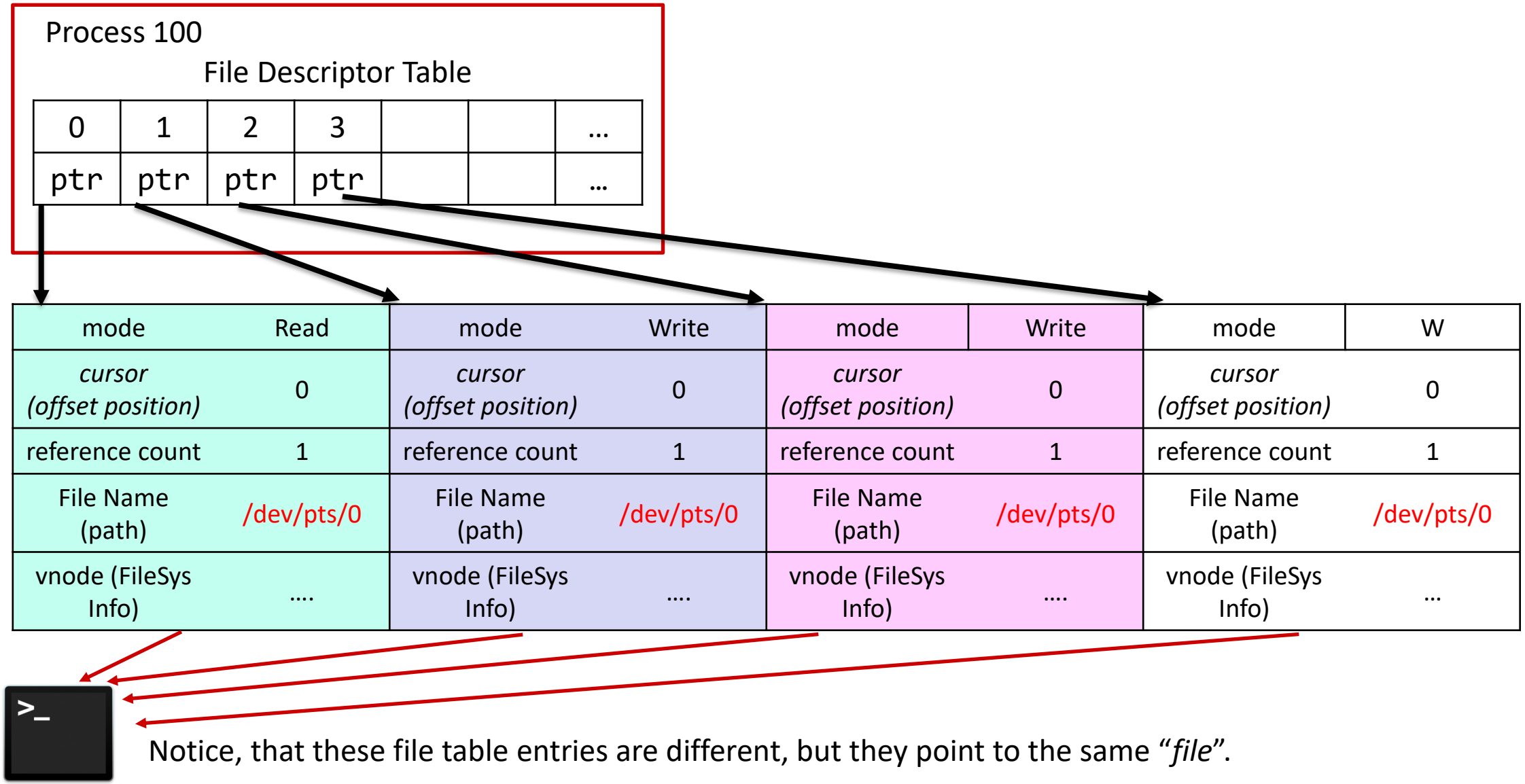


Let's go ahead and try to print to the terminal, without using STDOUT_FILENO!

The terminal itself is *treated as a file! Let's see what that file is called.*

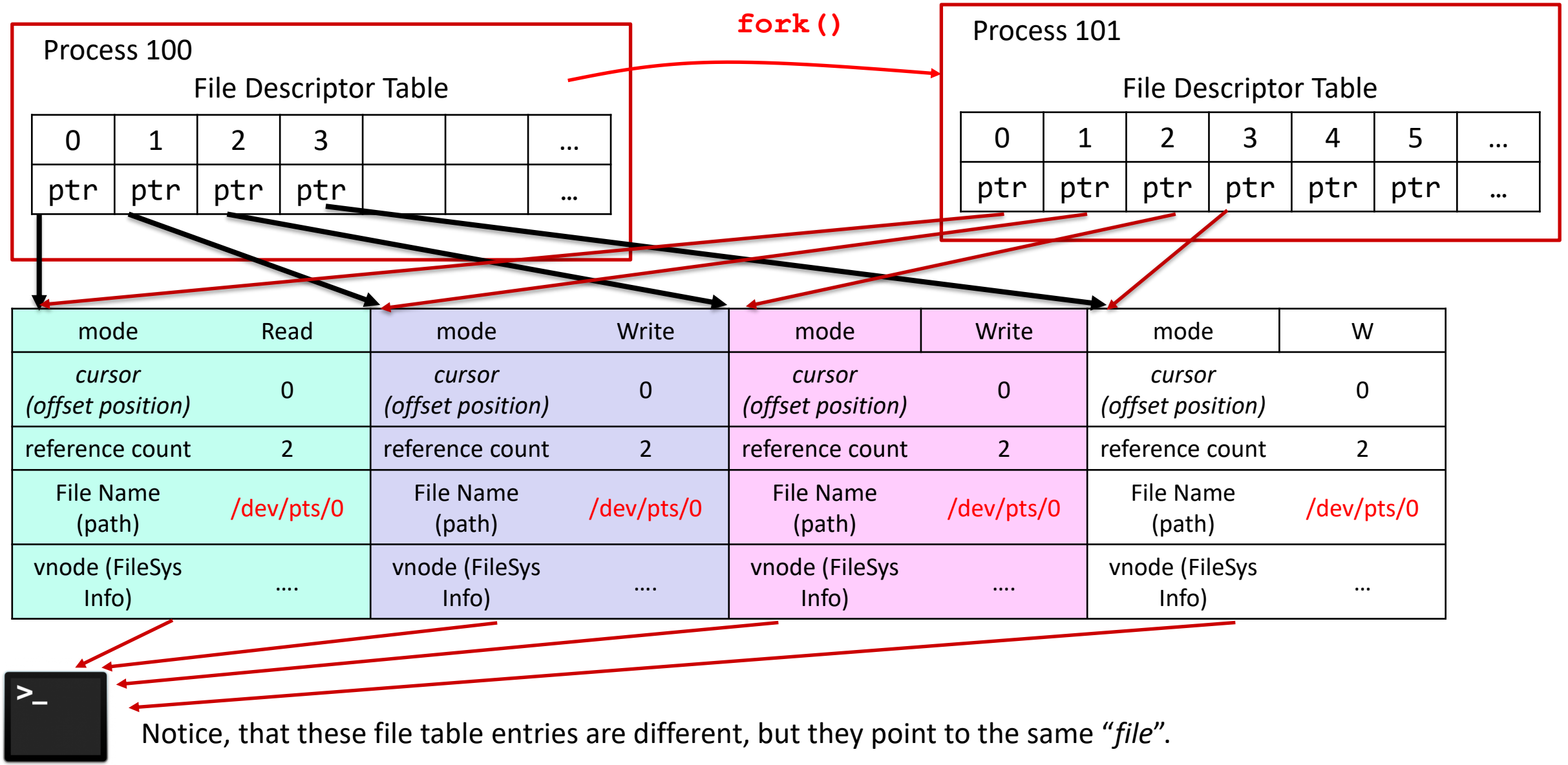
Notice, that these file table entries are different, but they point to the same "*file*".

Demo: Terminal Printing



Notice, that these file table entries are different, but they point to the same "*file*".

Open File Table With Fork!



Notice, that these file table entries are different, but they point to the same "*file*".

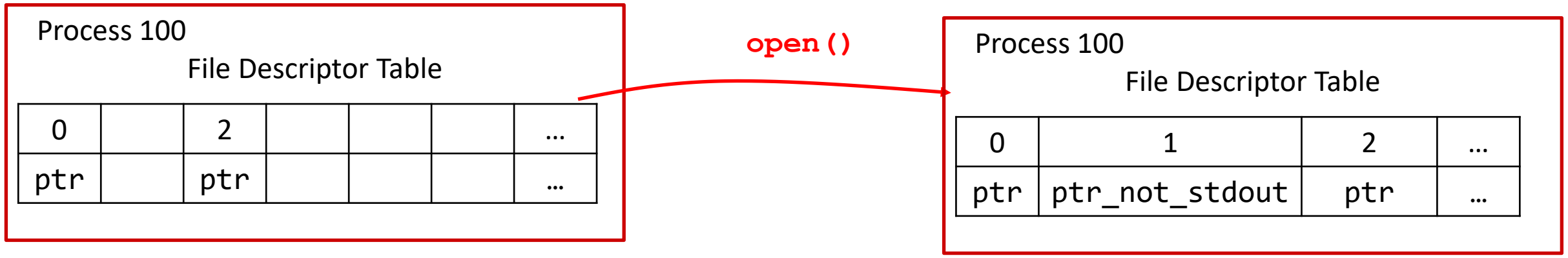
Lecture Outline

- wait(), waitpid(), WNOHANG (From Last Week)
- Intro to file descriptors
- File Descriptors
- Open File Table
- **Redirection & Pipes**
- Unix Commands & Controls

Redirecting File Descriptors

`printf` is implemented using `write(STDOUT_FILENO….)`
That's why it is redirected after changing stdout

- We can manipulate the File Table so that STDOUT_FILENO FD Table entry is associated with another file.
  - *Now, any writes to STDOUT_FILENO are redirected!*
- To do this without anything fancy, let's just close STDOUT_FILENO…



`Demo: close_stdout.c`

Redirecting stdin/out/err/everything

- We can manipulate the File Table so that a FD Table entry is associated with another file.
- ```
  int dup2(int oldfd, int newfd);
  ```
  - The file descriptor *newfd* is adjusted so that it now refers to the same open file description as *oldfd*. (newfd is closed silently…shh)

```
int dup2(int redirect_here, STDOUT_FILENO);
```

  - In this example, STDOUT_FILENO, no longer refers to the terminal, but rather the FILE associated with ***redirect_here***

Demo: dup_stdout.c
```

We all need dup2()

Process 100

File Descriptor Table

0	1	2	3			…
ptr	ptr	ptr	ptr			…

```
dup2(new_file_fd, STDOUT_FILENO);
```

new_file_fd = 3

mode	Read	mode	Write	mode	Write	mode	W
cursor (offset position)	0	*cursor (offset position)*	0	*cursor (offset position)*	0	*cursor (offset position)*	0
reference count	1	reference count	1	reference count	1	reference count	1
File Name (path)	/dev/pts/0	File Name (path)	/dev/pts/0	File Name (path)	/dev/pts/0	File Name (path)	my_file.txt
vnode (FileSys Info)	….	vnode (FileSys Info)	….	vnode (FileSys Info)	….	vnode (FileSys Info)	…

We all need dup2()

```
dup2(new_file_fd, STDOUT_FILENO);
```

This first closes STDOUT_FILENO, and because **ref_count is 0**, we remove it from the Open File Table

Then, it duplicates the entry to match new_file_fd

Process 100

File Descriptor Table

0	1	2	3			…
ptr		ptr	ptr			…

mode	Read
cursor (offset position)	0
reference count	1
File Name (path)	/dev/pts/0
vnode (FileSys Info)	….

Free space…

mode	Write
cursor (offset position)	0
reference count	1
File Name (path)	/dev/pts/0
vnode (FileSys Info)	….

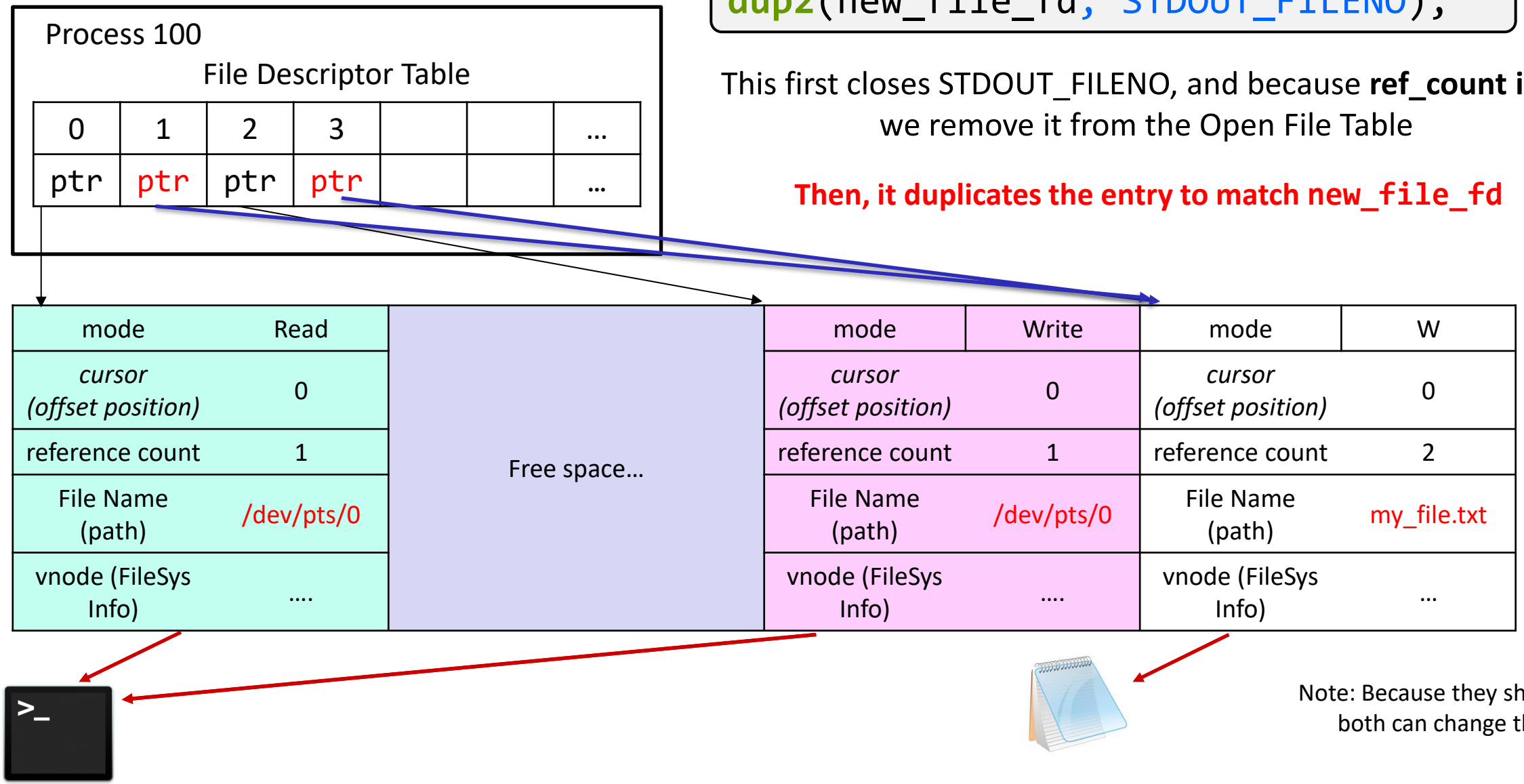
mode	W
cursor (offset position)	0
reference count	1
File Name (path)	my_file.txt
vnode (FileSys Info)	…

We all need dup2()

```
dup2(new_file_fd, STDOUT_FILENO);
```

This first closes STDOUT_FILENO, and because **ref_count is 0**, we remove it from the Open File Table

Then, it duplicates the entry to match new_file_fd



Note: Because they share this, they both can change the cursor.

Interprocess Communication: *Pipes*

```
int pipe(int pipefd[2]);
```

- Creates a unidirectional data channel for IPC
  - Communication through file descriptors! // POSIX ☺
- Takes in an array of two integers, and sets each integer to be a file descriptor corresponding to an "end" of the pipe
- `pipefd[0]` is the reading end of the pipe
- `pipefd[1]` is the writing end of the pipe

memorize: **you *read* before you write**
r is before w

Pipe Visualization

- A pipe "file" that has distinct file descriptors for reading and writing. This "file" only exists as long as there are references to it and is maintained by the OS.
  - Data written to the pipe is stored in a buffer until it is read from the pipe.

Process 100

File Descriptor Table

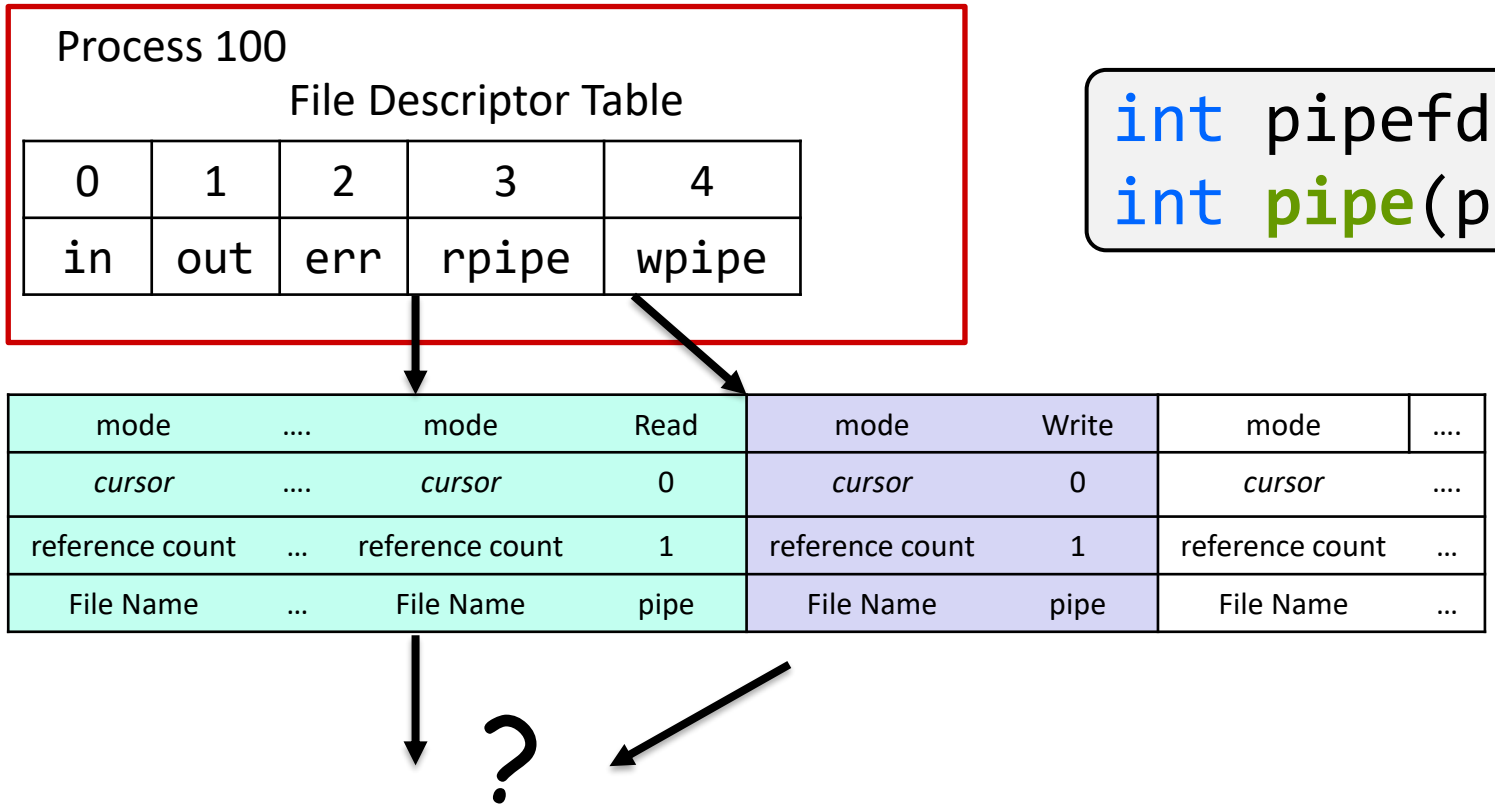
0	1	2
in	out	err

```
int pipefd[2];
int pipe(pipefd);
```

mode	....	mode	....	mode	....	mode	....
cursor	....	*cursor*	....	*cursor*	....	*cursor*	....
reference count	...	reference count	...	reference count	...	reference count	...
File Name	...	File Name	...	File Name	...	File Name	...

Pipe Visualization

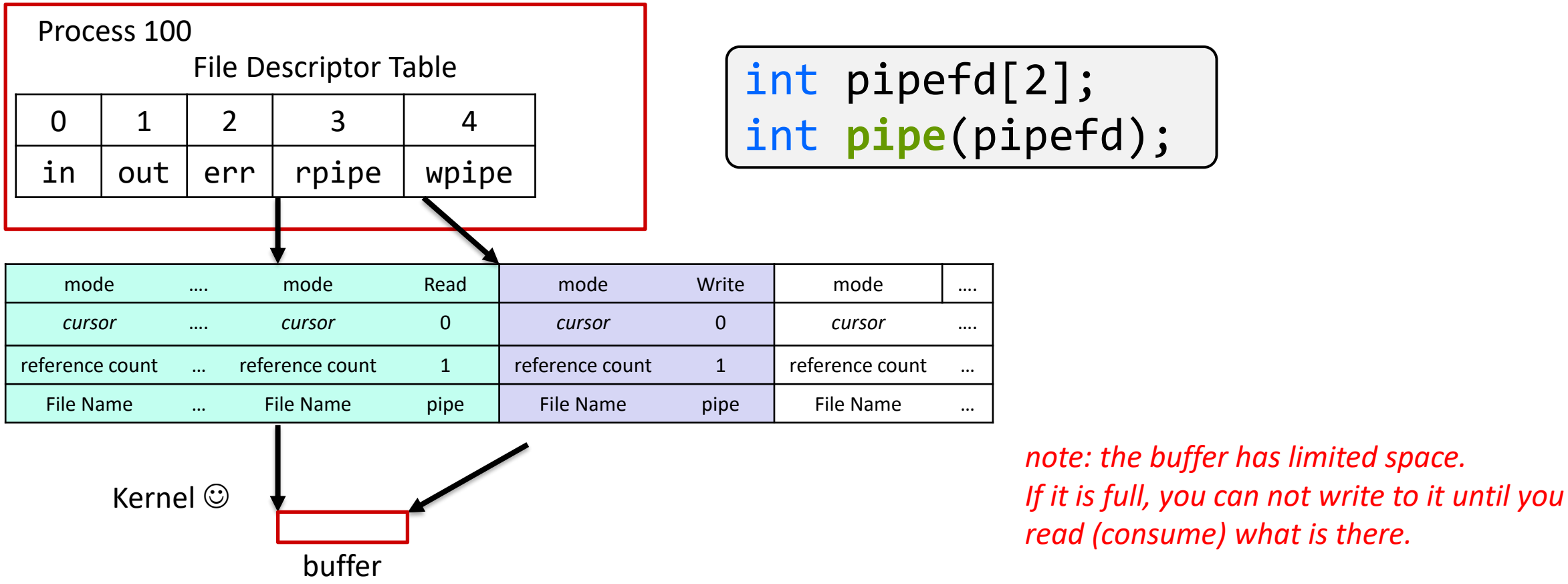
- Creating a pipe initializes two file descriptors in the process FD Table.
- Makes two entries in the system wide file table!




```
int pipefd[2];
int pipe(pipefd);
```

Pipe Visualization

- Creating a pipe initializes two file descriptors in the process FD Table.
- Makes two entries in the system wide file table!




```
int pipefd[2];
int pipe(pipefd);
```

note: the buffer has limited space.
If it is full, you can not write to it until you read (consume) what is there.

Poll Everywhere

pollev.com/tqm

- What does the parent print?
- What does the child print?

```
static char *message = "Hello!\n";
static char *p_message = "Bye!\n";

int main() {
  int pipe_fds[2];
  pipe(pipe_fds);
  pid_t pid = fork();
  if (pid == 0) {
    close(pipe_fds[0]); // Which end of the pipe is this?
    write(pipe_fds[1], message, strlen(message) + 1);
    char str[strlen(p_message) + 1];
    ssize_t chars_read = read(pipes_fd[1], str, sizeof(str));
    if (chars_read != -1) {
      printf("%s", str);
    }
    return EXIT_SUCCESS;
  }
  close(pipe_fds[1]); // Which end of the pipe is this?
  char str[strlen(message) + 1];
  ssize_t chars_read = read(pipe_fds[0], str, sizeof(str));
  if (chars_read != -1) {
    printf("%s", str);
  }
  write(pipe_fds[0], p_message, strlen(p_message) + 1);
  return EXIT_SUCCESS;
}
```

Poll Everywhere

Discuss

- What is the behavior of this program? Does read fail?

```c
int main(){

    int pipe_fds[2];
    pipe(pipe_fds);

    char str[10];
    ssize_t chars_read = read(pipe_fds[0], str, sizeof(str));

    return EXIT_SUCCESS;
}
```

pipe_wait.c

Pipes & EOF

- When ***reading,*** you read until you've read enough bytes or EOF.
- ***When using a pipe, if there is nothing in the buffer to read, then you will wait until there is.***
  - ***EOF will not be returned nor will 0, when there is nothing in the pipes buffer.***
  - ***You must write to the write end of the pipe for the corresponding read to return!***

- EOF is only read from a pipe when:
  - All write ends of the pipe are closed, it's impossible to read anything from there.
  - Ask yourself, how can a child indicate to its parent that it is done writing? :-)

- **This will cause many bugs in your programs. Make sure to always close the FD you no longer need. *Even one reference to the Write end of the pipe will cause all READS to block.***

Lecture Outline

- wait(), waitpid(), WNOHANG (From Last Week)
- Intro to file descriptors
- File Descriptors
- Open File Table
- Redirection & Pipes
- **Unix Commands & Controls**

Unix Shell

- A **user level** process that reads in commands
  - This is the terminal you use to compile, and run your code
- Commands can either specify one of our programs to run or specify one of the already installed programs
  - Other programs can be installed easily
- There are many commonly used bash programs, we will go over a few and other important bash things

Path Specifiers

- `/` is used to connect directory and file names together to create a file path
  - E.g. **`workspace/5480/penn-shell/`**
- `.` is used to specify the current directory
  - E.g. **`./test_suite`** tells to look in the current directory for a file called **`test_suite`**
- `..` is like `.` but refers to the parent directory
  - E.g. **`./solution_binaries/../test_suite`** would be effectively the same as the previous example

Common Commands (Pt. 1)

- **`ls`** lists out the entries in the specified directory (or current directory if another directory is not specified)

- **`cd`** changes directory to the specified directory
  - E.g. **`cd ./solution_binaries`**

- **`exit`** closes the terminal

- **`mkdir`** creates a directory of specified name

- **`touch`** creates a specified file or updates the time stamp if it already exists

Common Commands (Pt. 2)

- **`echo`** takes in command line args and simply prints those args to `stdout`
  - **`echo hello!`** simply prints **`hello!`**

- **`wc`** reads a file or from `stdin` and prints out the line count, word count, and byte count of the input

- **`cat`** prints out the contents of a specified file to `stdout`. If no file is specified, prints out what is read from `stdin`

- **`head`** print the first 10 line of specified file or `stdin` to `stdout`

Common Commands (Pt. 3)

- **`grep`** given a pattern (regular expression) searches for all occurrences of such a pattern. Can search a file, search a directory recursively or `stdin`. Results printed to `stdout`
- **`history`** prints out the history of commands used by you on the terminal
- **`cron`** a program that regularly checks for and runs any commands that are scheduled via "crontab"
- **`wget`** specify a URL, and it will download that file for you

Unix Shell Commands

- Commands can also specify flags
  - E.g. `ls -l` lists the files in the specified directory in a more verbose format

- Revisiting the design philosophy:
  - Programs should "Do One Thing And Do It Well."
  - Programs should be written to work together
  - Write programs that handle text streams, since text streams is a universal interface.

- <u>These programs can be easily combined with UNIX Shell operators to solve more interesting problems</u>

Unix Shell Control Operators

- **`cmd1 && cmd2`** used to run two commands. The second is only run if **`cmd1`** doesn't fail
  - E.g. **`make && ./test_suite`**

- **`cmd1 | cmd2`** creates a pipe so that the `stdout` of **`cmd1`** is *redirected* to the `stdin` of **`cmd2`**
  - E.g. **`history | grep valgrind`**

- **`cmd &`** runs the process in the background, allowing you to immediately input a new command

Unix Shell Control Operators

- **`cmd < file`** *redirects* `stdin` to instead read from the specified file
  - E.g. **`./penn-shell < script.txt`**

- **`cmd > file`** redirects the `stdout` of a command to be written to the specified file
  - E.g. **`grep -r kill > out.txt`**

- Complex example:

```
cat ./input.txt | ./numbers > out.txt
&& diff out.txt expected.txt
```

Poll Everywhere

pollev.com/tqm

- Which of the following commands will print the number of files in the current directory?

A. `ls > wc`
B. `cd . && ls wc`
C. `ls | wc`
D. `ls && wc`
E. **The correct answer is not listed**
F. **We're lost…**

cd: change directory

ls: list directory contents

wc: reads from stdin, prints the number of words, lines, and characters read.

Poll Everywhere

pollev.com/tqm

- Is this valid?
- `ls | sort -r | cat | cat | cat | cat`
  - sort sorts the input in alphabetical order, -r is in reverse order.

Poll Everywhere

Discuss

- If there's time, how would we even implement this?
  - `ls | sort -r`
- How many processes are necessary, do we need pipes, what about dup2?

That's All!

- See you next time :)