Pipes & File Descriptors

Computer Operating Systems, Spring 2024

Instructors: Joel Ramirez Travis McGaha

Head TAs: Ash Fujiyama Emily Shen Maya Huizar

TAs:

Ahmed Abdellah	Bo Sun	Joy Liu	Susan Zhang	Zihao Zhou
Akash Kaukuntla	Connor Cummings	Khush Gupta	Vedansh Goenka	
Alexander Cho	Eric Zou	Kyrie Dowling	Vivi Li	
Alicia Sun	Haoyun Qin	Rafael Sakamoto	Yousef AlRabiah	
August Fu	Jonathan Hong	Sarah Zhang	Yu Cao	

Poll Everywhere

pollev.com/cis5480

- How confident do you feel in your penn-parser? Would you feel comfortable expanding on it slightly and/or using it in penn-shell (the next homework to release)?

Administrivia

- Project 0/1 penn-vec & penn-parser:
  - Due this Friday at Midnight!
  - You can use late tokens; 1 token is 2 days.

Administrivia

- There will be a check-in due next week
  - Due before Tues @ 12pm

- First recitation is today, there will be another again next week
  - Thursday @ 7PM in AGH!
    - Next week who know's where it'll be

Administrivia

- Penn-shell partner info will be up soon; will make an announcement on Ed
  - Project 1 is done in pairs
  - You will also have to review each other's code from Project 0 and give feedback
  - People without partners will be randomly assigned

Lecture Outline

- **Signal Blocking & sigsuspend**
- Intro to file descriptors
- File Descriptors: Big picture
- Redirection & Pipes
- Unix Commands & Controls

Signal Blocking

- A process maintains a set of signals called a "signal mask"
  - Signals in that set/mask are "**blocked**"
  - **Signals that are "blocked" are delayed in being delivered to the process**, once unblocked, the process responds to the signals accordingly according to the corresponding disposition.
  - Signals are added to a "pending set" of signals to be delivered once unblocked.

- ***This is not the same as ignoring a signal.***

```
struct sigaction sa = {0};
sa.sa_handler = SIG_IGN;
sa.sa_flags = SA_RESTART;
sigaction(SIGNAL, &sa, NULL);
```

When you set a signal's disposition to SIG_IGN, then when a process receives the signal it simply throws it away.

- **Reminder: Process Blocked != Signals are Blocked**

sigset_t

sigset_t types must be initialized by a call to **sigemptyset()** when used with a number of different sigsetops. IF NOT THE BEHAVIOR IS UNDEFINED. ☺

- `sigset_t` is a typedef'd bitset to maintain the set of signals blocked
- `int sigemptyset(sigset_t* set);`
  - initializes a sigset_t to be empty
- `int sigaddset(sigset_t* set, int signum);`
  - Adds a signal to the specified signal set
- More functions & details in man pages
  - (`man sigemptyset`)
- Example snippet:

```
sigset_t mask;
if (sigemptyset(&mask) == -1) {
  // error
}
if (sigaddset(&mask, SIGINT) == -1) {
  // error
}
```

sigprocmask()

```
int sigprocmask(int how, const sigset_t* set, sigset_t* oldset);
```

- Sets the process mask to be the specified process mask, set, depending on the value of `int` how
  - "how would you like me to use set?"
- `const sigset_t* set`
  - Is the `set` you would like you use with how
- `sigset_t* oldset`
  - Is set to the previous value of the signal mask.

sigprocmask()

```
int sigprocmask(int how, const sigset_t* set, sigset_t* oldset);
```

- Sets the process mask to be the specified process “block” mask
- `int how`
  - `SIG_BLOCK`
    - The new mask is the union of the current mask and the specified set.
  - `SIG_UNBLOCK`
    - The new mask is the intersection of the current mask and the complement of the specified set.
  - `SIG_SETMASK`
    - The current mask is replaced by the specified set.

Recall: Buggy Code with Critical Section

```c
// assume this works
void list_push(list* this, float f) {
  Node* node = malloc(sizeof(Node));
  if (node == NULL) {
    exit(EXIT_FAILURE);
  }
  node->value = f;
  node->next = NULL;
  this->tail->next = node;
  this->tail = node;
}
```

```c
void handler(int signo){
    list_push(list, 4.48);
}
int main(int argc, char *argv[]){
    //sa setup omitted, handler set, etc.
    sigaction(SIGINT, &sa, NULL);

    float f;
    while (list_size(list) < 20){
     read_float(stdin, &f);
     list_push(list, f);
    }
}
```

`list_push` can be called from two places.

Either from the *main* or from the *signal handler*

While the process is wrangling the values for the linked-list, the process could be interrupted and forced to run the signal handler.

Note: This refers to the English definition of "interrupt," not the computer science term.

Recall: Buggy Code with Critical Section

```
// assume this works
void list_push(list* this, float f) {
  Node* node = malloc(sizeof(Node));
  if (node == NULL) {
    exit(EXIT_FAILURE);
  }
  node->value = f;
  node->next = NULL;
  this->tail->next = node;
  this->tail = node;
}
```

Which portion of the code ***can safely have its execution paused*** so the signal handler can run before the original code resumes?

Ask yourself: where is *global information being modified?*

Poll Everywhere

pollev.com/cis5480

```c
// assume this works
void list_push(list* this, float f) {
  Node* node = malloc(sizeof(Node));
  if (node == NULL) {
    exit(EXIT_FAILURE);
  }
  node->value = f;
  node->next = NULL;
  this->tail->next = node;
  this->tail = node;
}
```

- *How can we make this function signal safe?*

sigsuspend()

- Instead of busy waiting and wasting CPU cycles (that can be used by other processes), we can suspend process execution instead.

- ```
  int sigsuspend(const sigset_t* mask);
  ```
  - Temporarily replaces process mask with specified one and suspends execution untill a signal that is not blocked is delivered.
  - If signal that is not blocked is received, the process *'returns'* from **sigsuspend**
    - The mask in place before the suspend call is restored.
    - If the signal received terminates the program, then the process never 'returns' from **sigsuspend.**
- Demo: `suspend_sigint.c`
  - Compare to previous code: `delay_sigint.c`
  - Less CPU resources used ☺
  ```

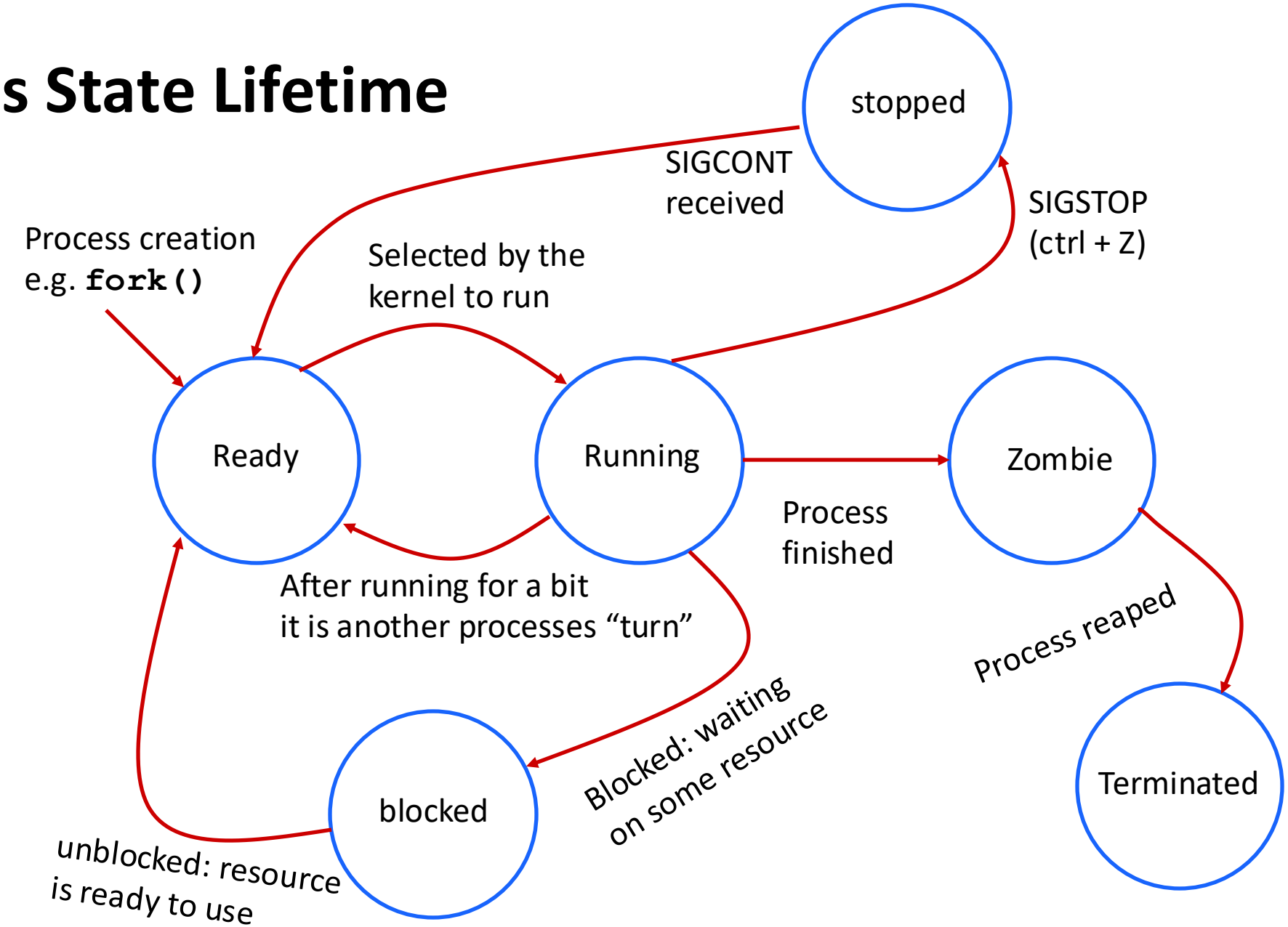
`volatile sig_atomic_t`

- If you need to communicate with a signal handler, we have been using global variables...
  - Modifying global variables is generally unsafe in signals.

- In "real world" code if you want to modify shared data within a signal handler, you should use global variable type: `volatile sig_atomic_t`
  - `volatile sig_atomic_t` is an integer type with interesting properties.

- We will not enforce this in these projects, but we felt like it was worth letting you know.

Process State Lifetime

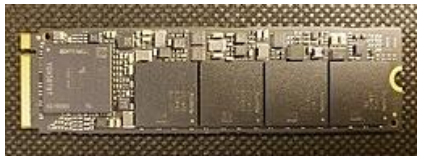


Lecture Outline

- Signal Blocking & sigsuspend
- **Intro to file descriptors**
- File Descriptors: Big picture
- 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()`**

stdout, stdin, 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
  - `stdout`: the normal terminal output. (buffered)
    - `stdout` in `C stdio.h`
    - `System.out` in Java
  - `stderr`: the terminal output for printing errors (unbuffered)
    - `stderr` in `C stdio.h`
    - `System.err` in Java

stdout, stdin, stderr

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

C unistd.h ×

Library > Developer > CommandLineTools > SDKs > MacOSX.sdk > usr > include > C 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 open on default for a process

- Printing to `stdout` 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 (won't dilly dally on this)
  - specify the set of permissions allowed in interacting with file
    - E.g. is the file, executable?, who can see it?, etc.
- Returns:
  - a valid file descriptor or -1 on error

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;
}
```

This enforces that a file isn't just created, but created *for the first time.*

Please check out the man page for open.

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

    // Do some magic with the file descriptor

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

Additional System FD Calls

```
ssize_t read(int fd, void *buf, 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, void *buf, size_t count);
```

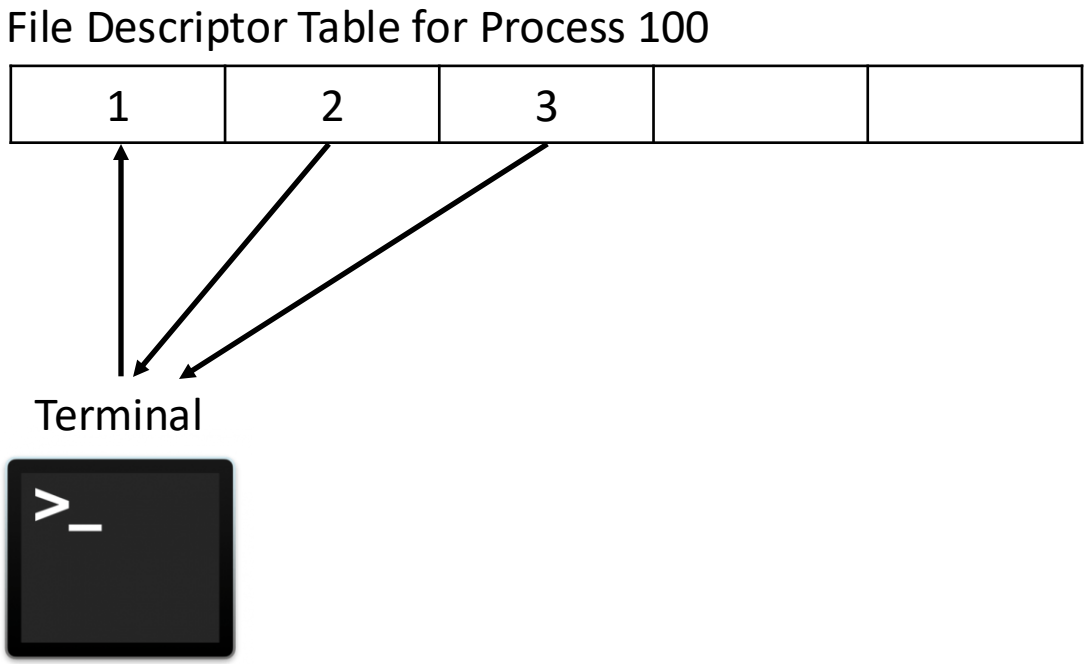
- writes up to *count* bytes from the buffer starting at *buf* to the file referred to by the file descriptor *fd*.
  - On success, the number of bytes written is returned. On error, -1 is returned, and *errno* is set to indicate the error.

Lecture Outline

- Intro to file descriptors
- **File Descriptors: Big Picture**
- Redirection & Pipes
- Unix Commands & Controls

File Descriptor Table

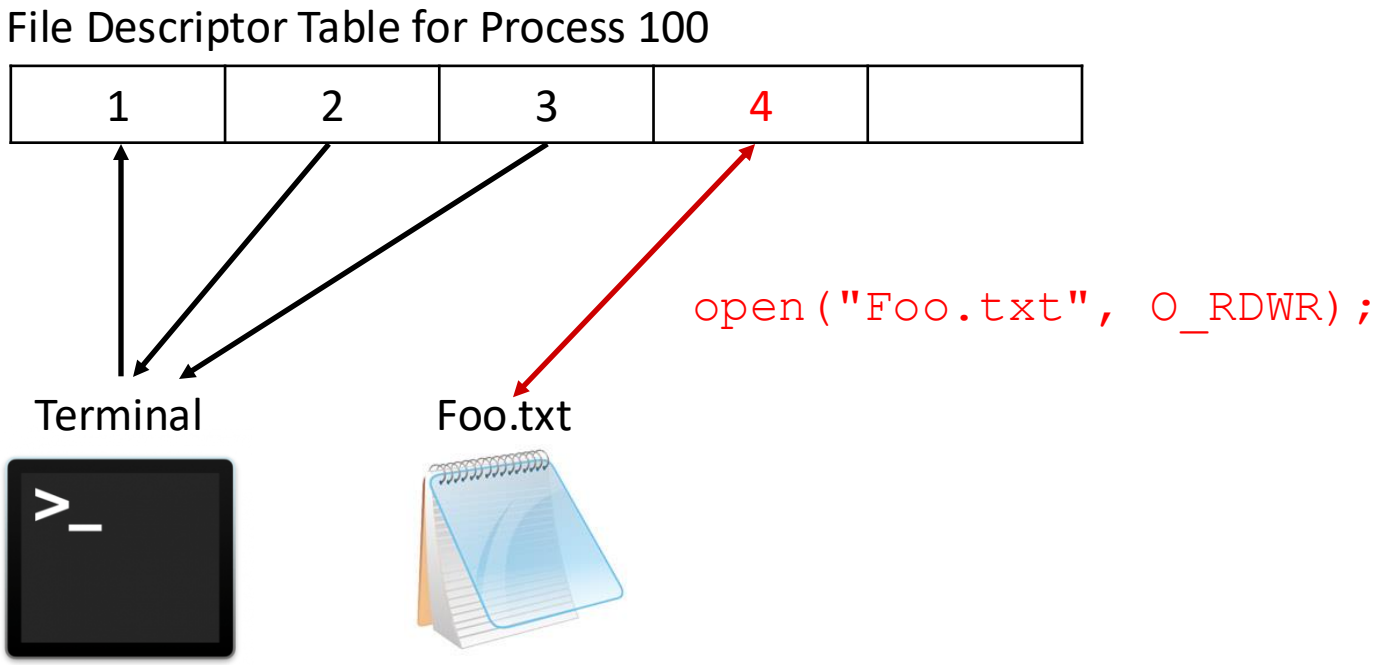
- Each process has **its own file descriptor table** managed by the OS
  - The table maintains information about the respective files the process has references to.
- A *file descriptor* is an index into a processes FD table.



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.

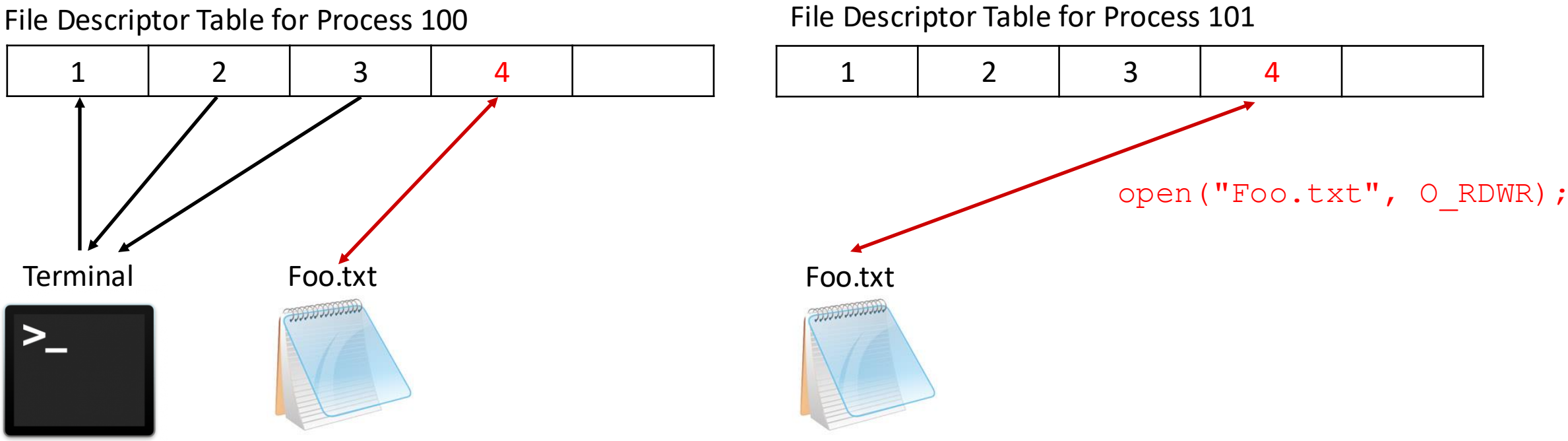


Not an accurate depiction.

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.



Not an accurate depiction.

Note: the lowest # fd is given when possible.

pollev.com/cis5480

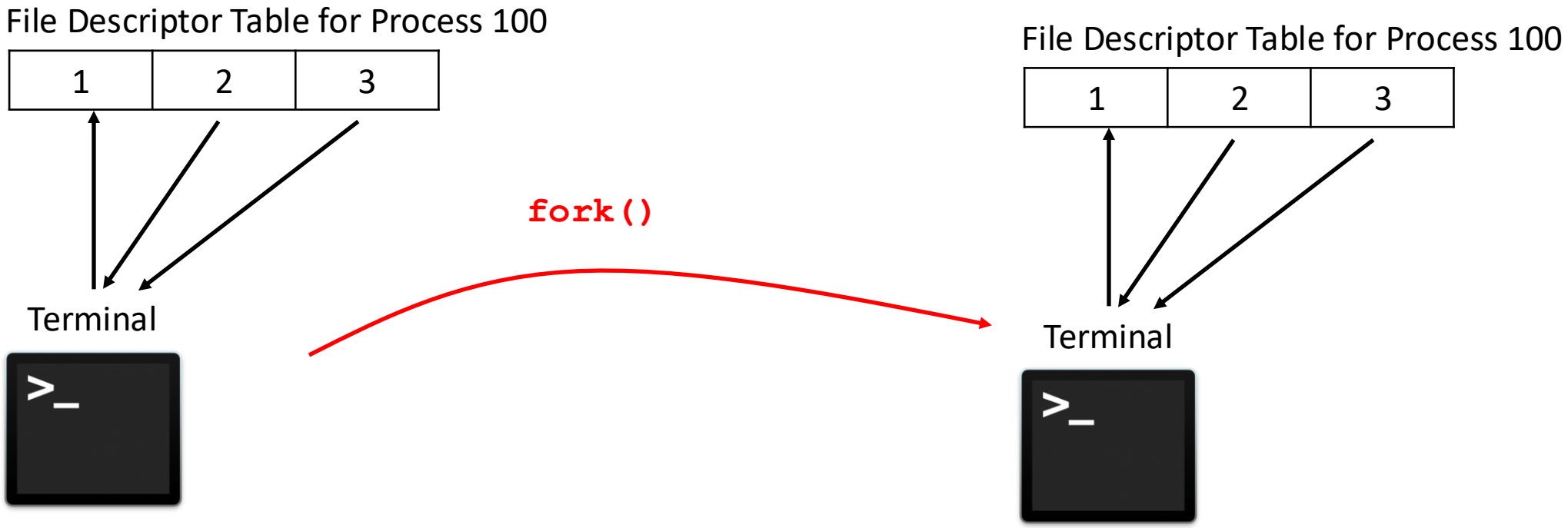
- What if there was only one global file descriptor table? What negative affects may this have?

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

1	2	3	4	

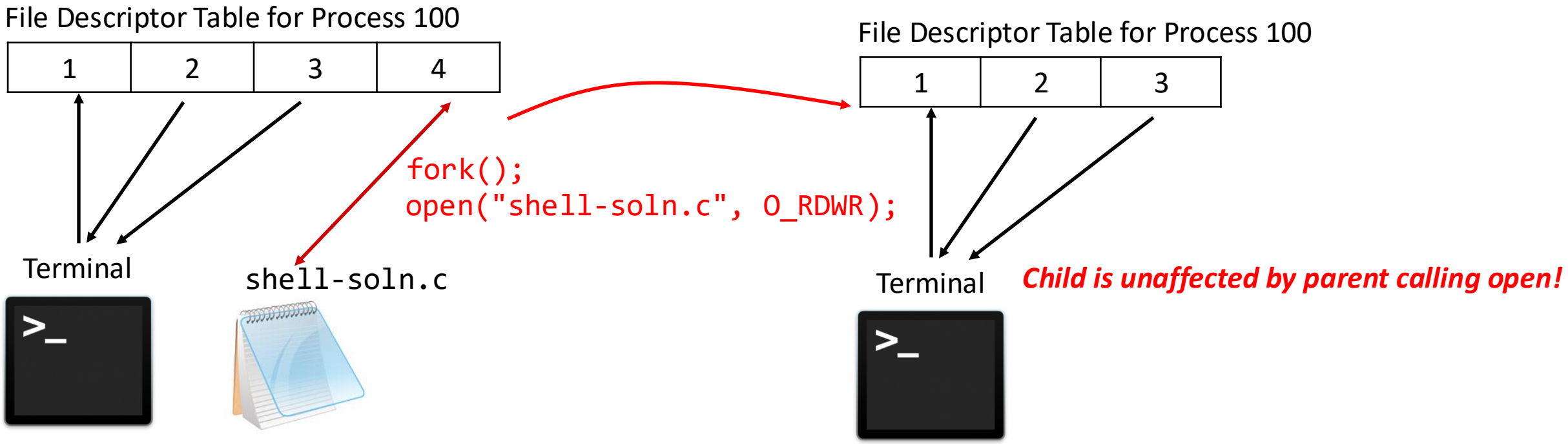
File Descriptor Table: Per Process

- 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



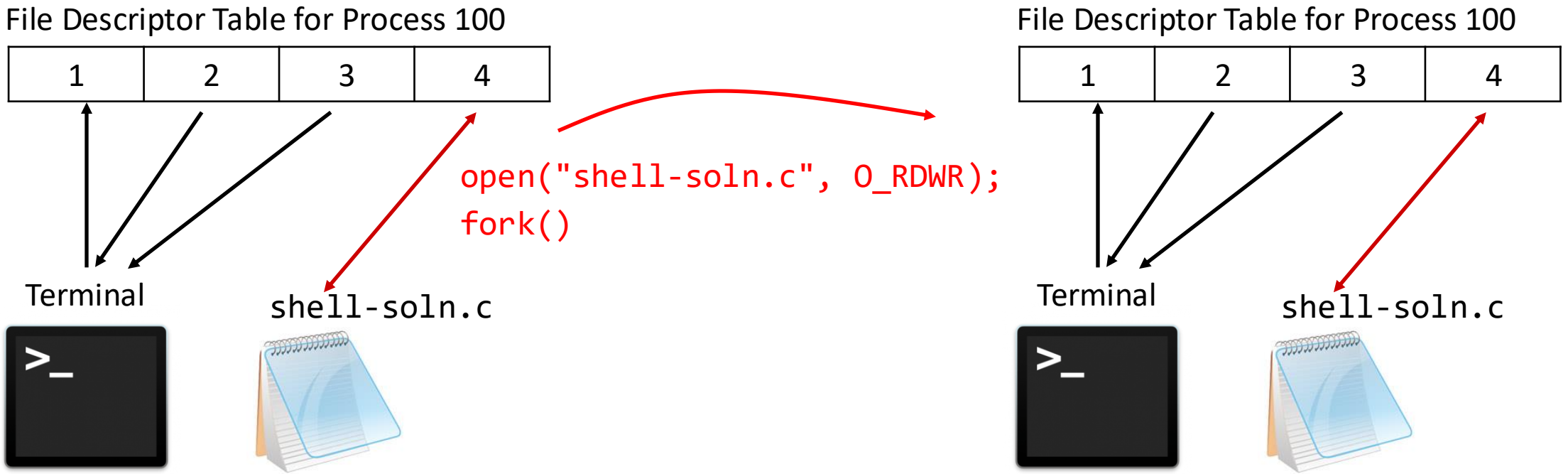
File Descriptor Table: Per Process

- 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***



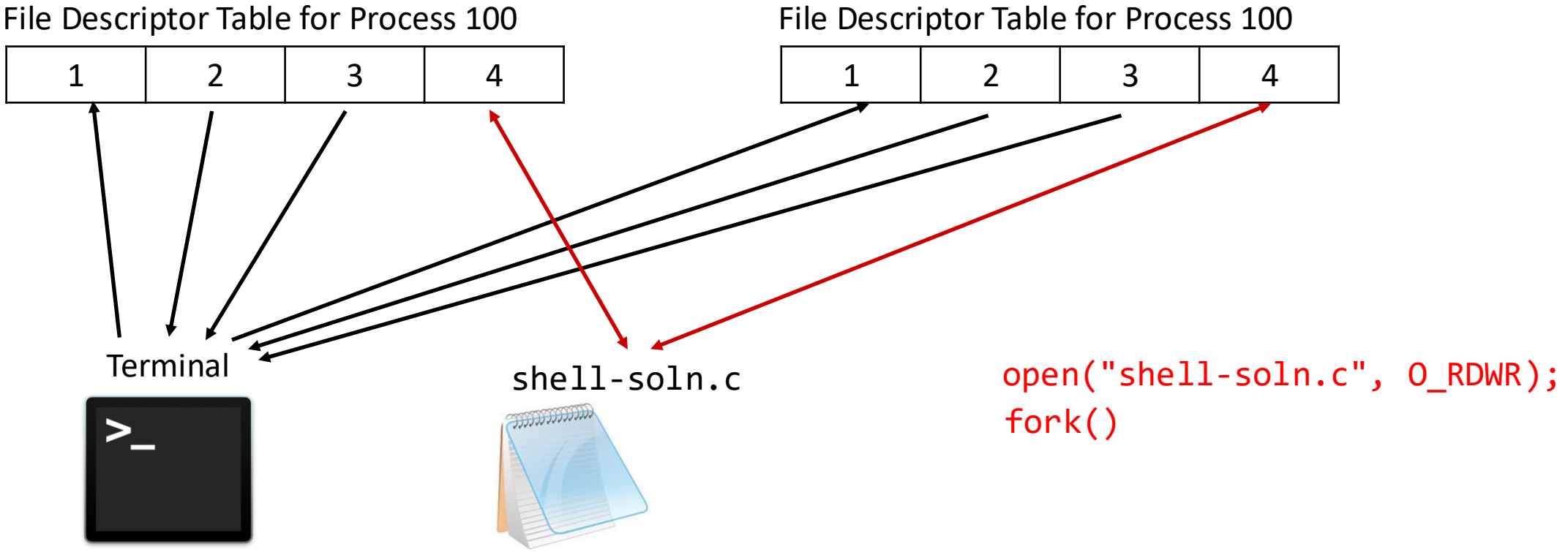
File Descriptor Table: Per Process

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



File Descriptor Table: Per Process

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



File Descriptor Table in Reality…

- 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!

As we open up more files, using open(), not only do we received a FD but an entry is made in the 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_a.txt	File Name (path)	File_b.txt	File Name (path)	…
vnode (FileSys Info)	….	vnode (FileSys Info)	….	vnode (FileSys Info)	….	vnode (FileSys Info)	…

File Descriptor Table in Reality…

- 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

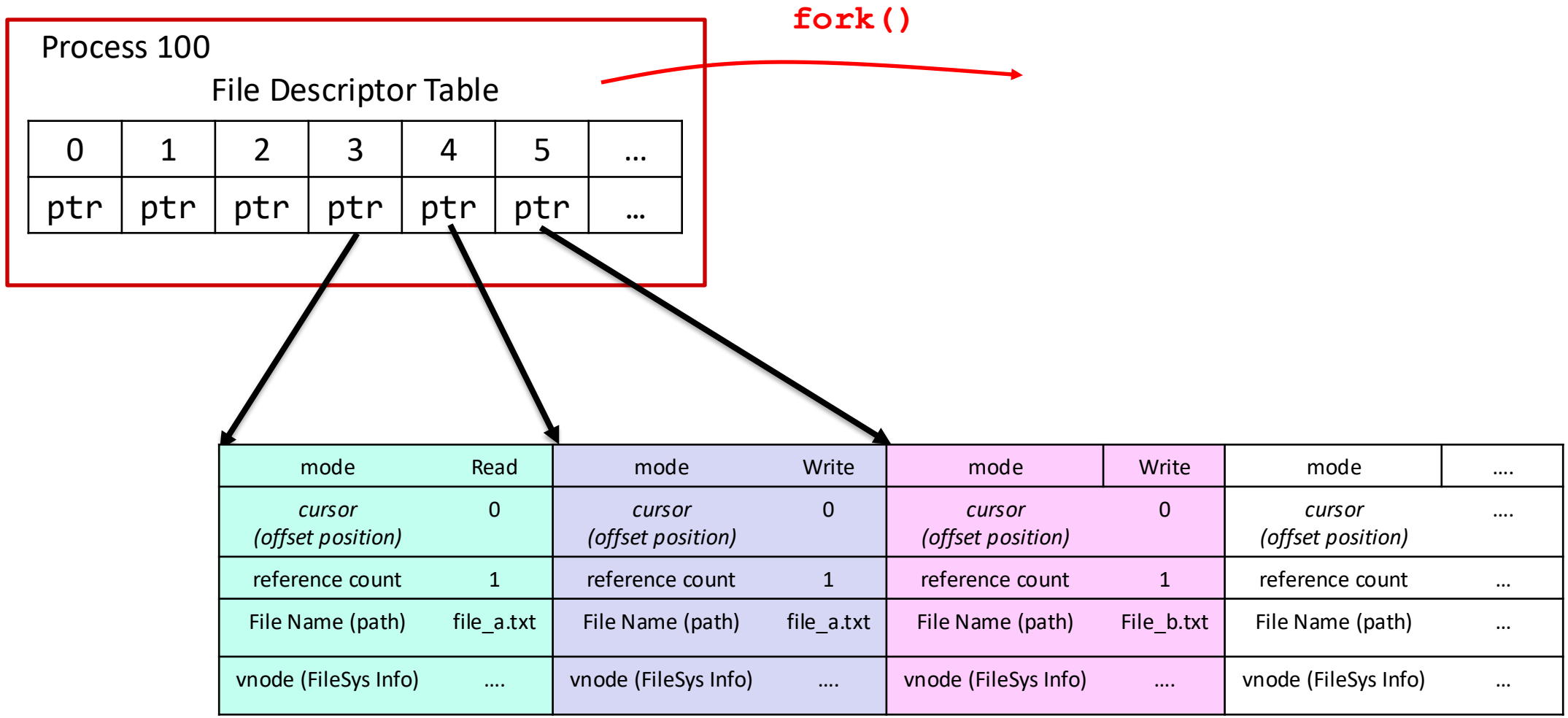
vnode/inode: don't worry yet….

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_a.txt	File Name (path)	File_b.txt	File Name (path)	…
vnode (FileSys Info)	….	vnode (FileSys Info)	….	vnode (FileSys Info)	….	vnode (FileSys Info)	…

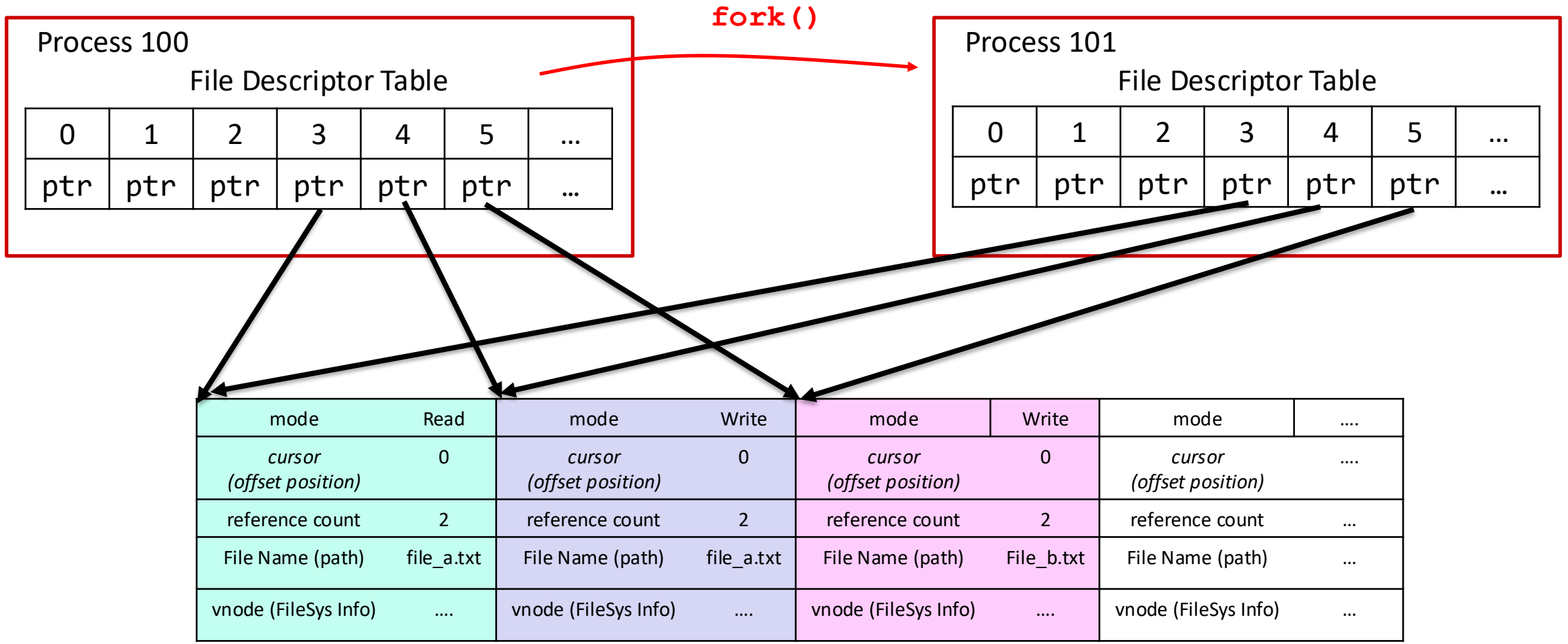
File Descriptor Table and Fork

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

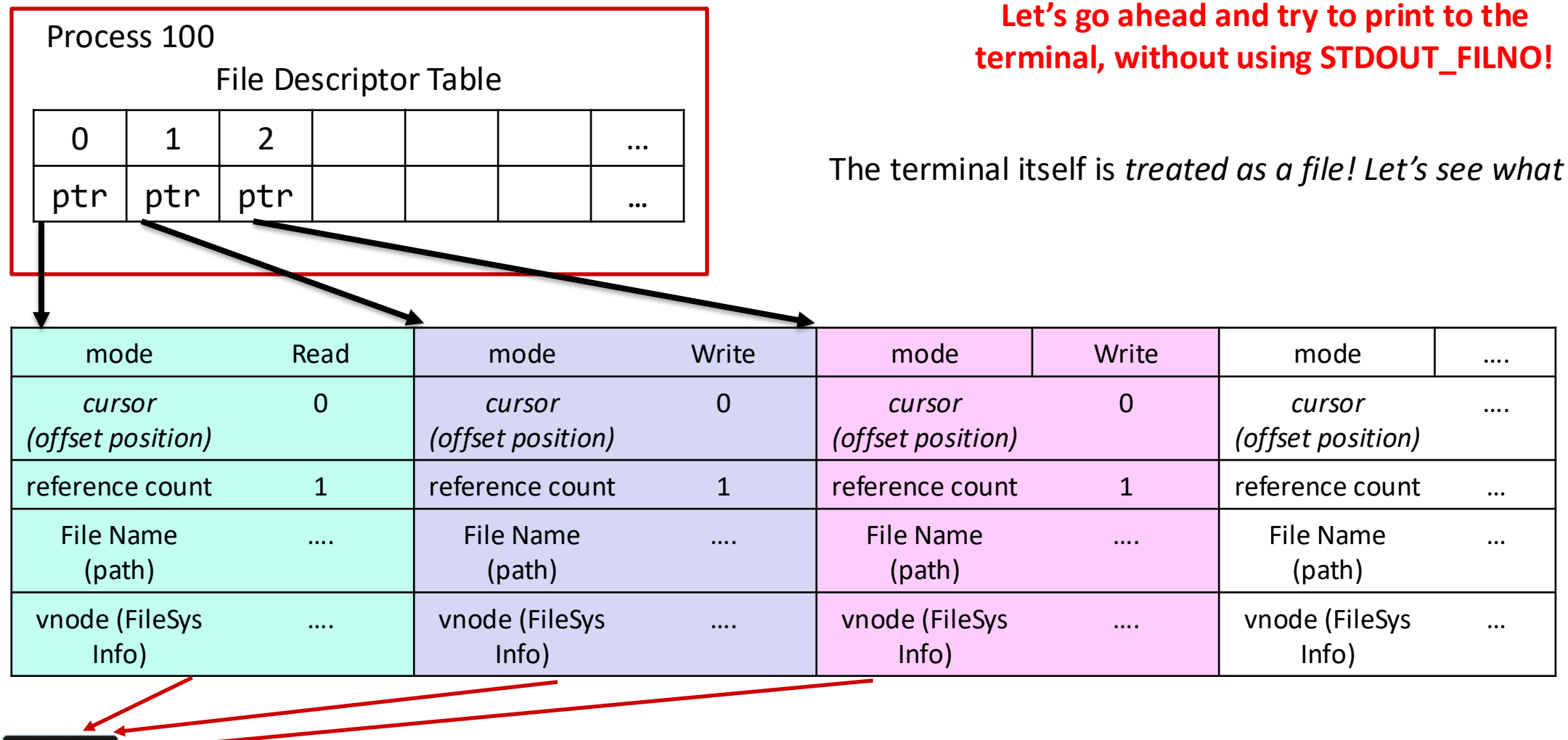


File Descriptor Table and Fork

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



Terminal Printing Demo

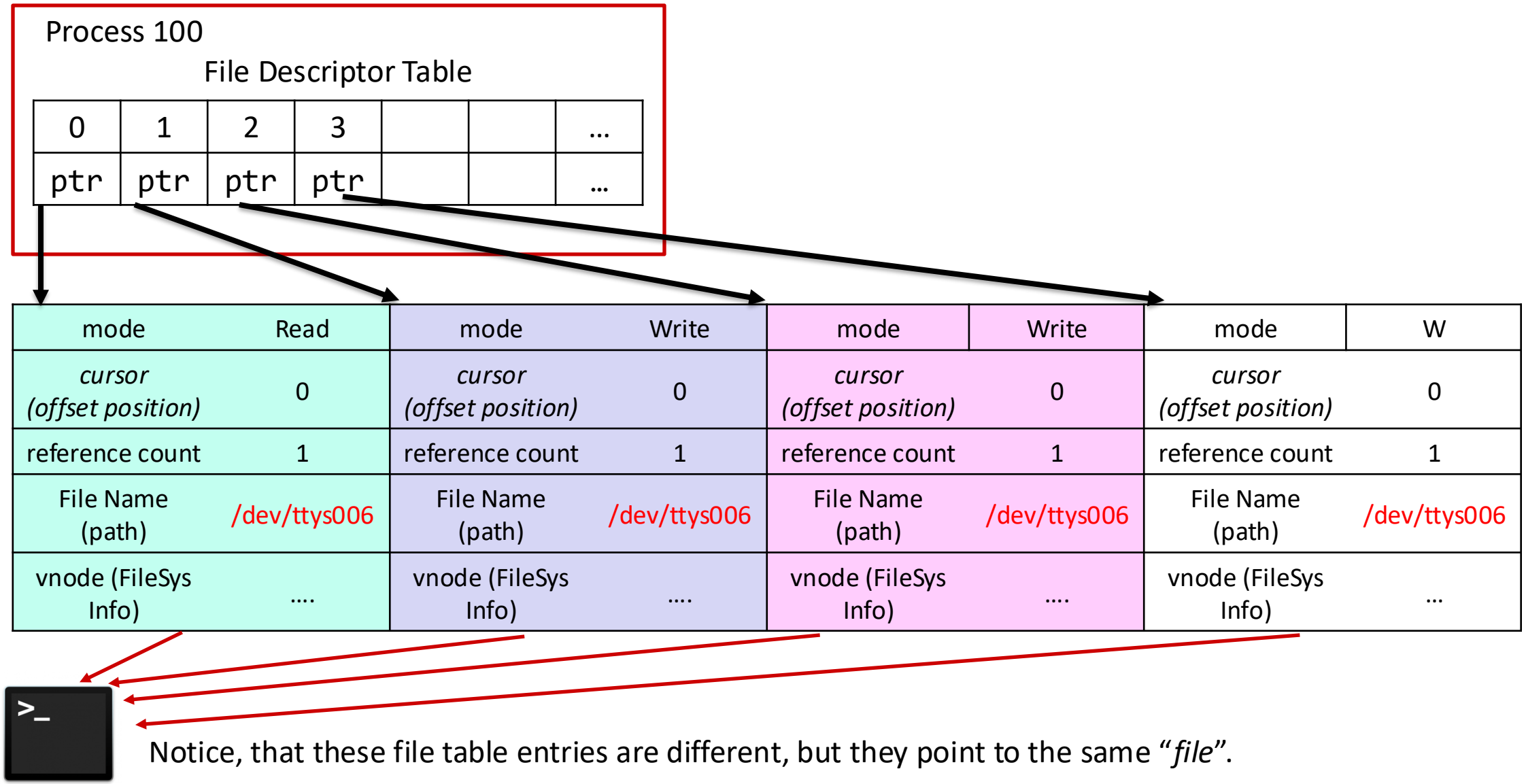


Let's go ahead and try to print to the terminal, without using STDOUT_FILNO!

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

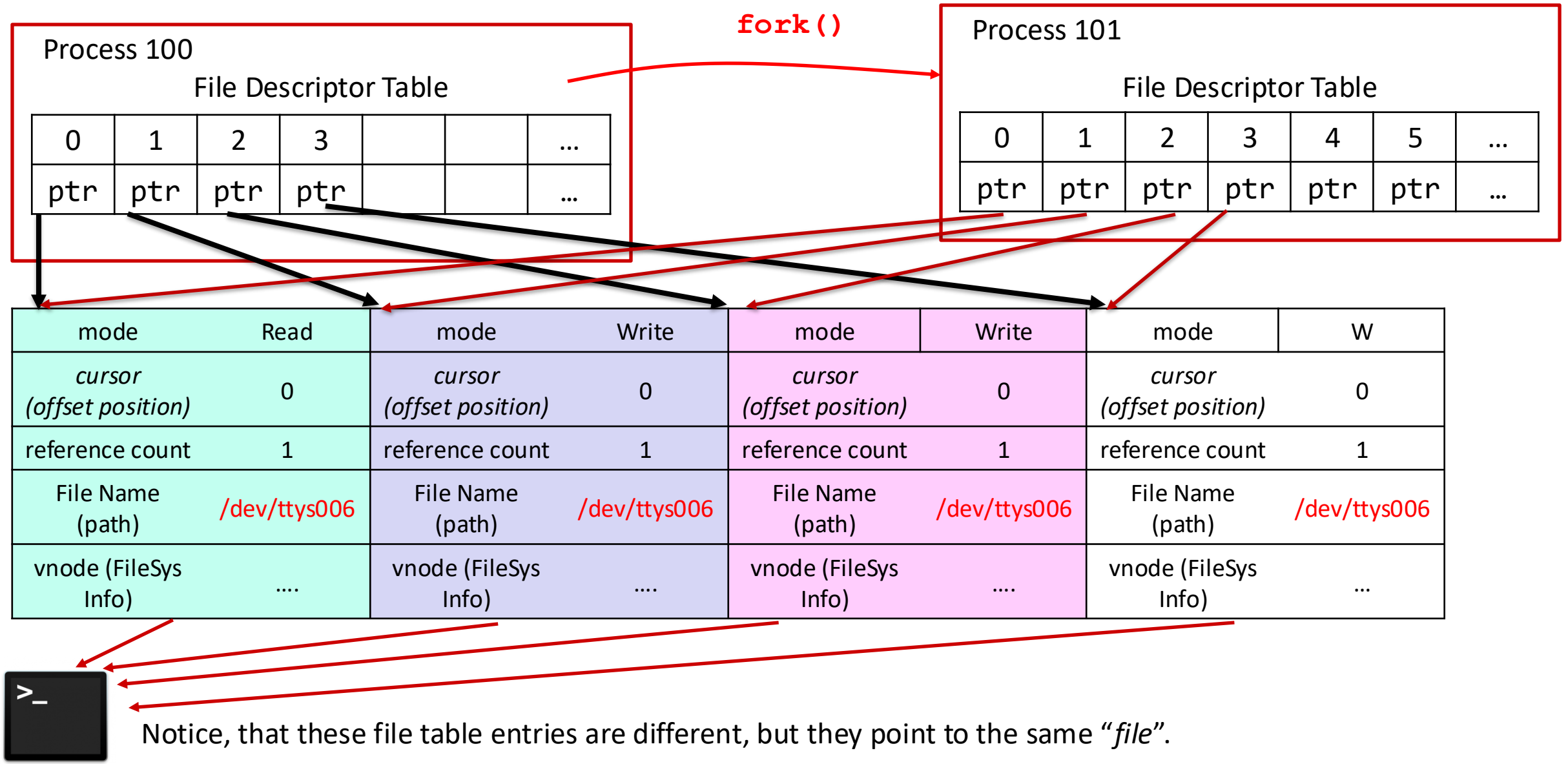
Notice, that these file table entries are different, but they point to the same "*file*".

Terminal Printing Demo



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

Terminal Printing Demo With Fork!



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

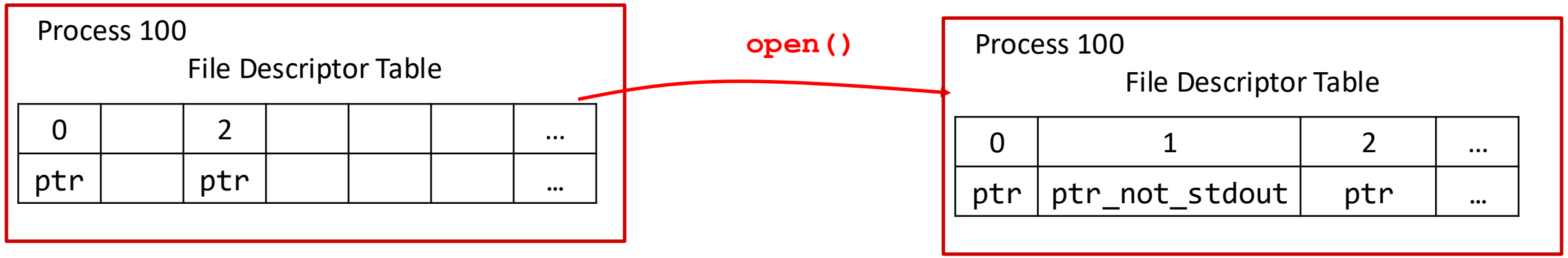
Lecture Outline

- Intro to file descriptors
- File Descriptors: Big Picture
- **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 referes 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);
```

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/ttys006	File Name (path)	/dev/ttys006	File Name (path)	/dev/ttys006	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/ttys006
vnode (FileSys Info)	….

Free space…

mode	Write
cursor (offset position)	0
reference count	1
File Name (path)	/dev/ttys006
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

Process 100

File Descriptor Table

0	1	2	3			…
ptr	ptr	ptr	ptr			…

mode	Read
cursor (offset position)	0
reference count	1
File Name (path)	/dev/ttys006
vnode (FileSys Info)	….

Free space…

mode	Write
cursor (offset position)	0
reference count	1
File Name (path)	/dev/ttys006
vnode (FileSys Info)	….

mode	W
cursor (offset position)	0
reference count	2
File Name (path)	my_file.txt
vnode (FileSys Info)	…

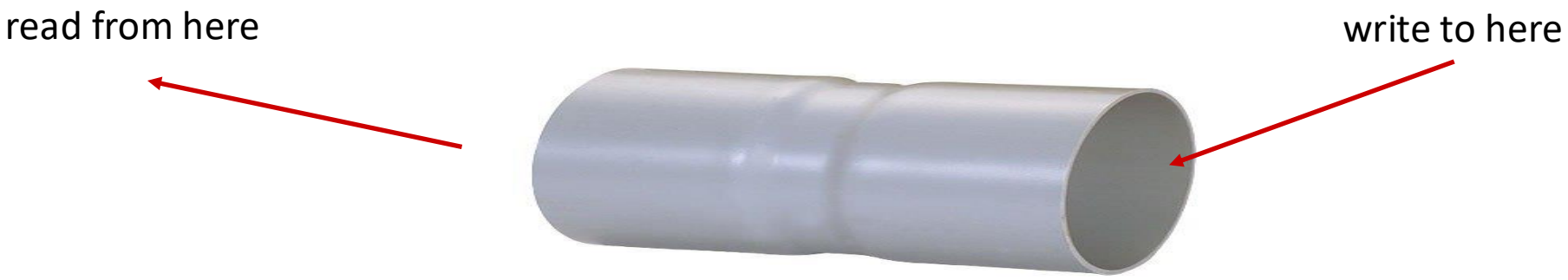
Note: Because they share this, they both can change the cursor.

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

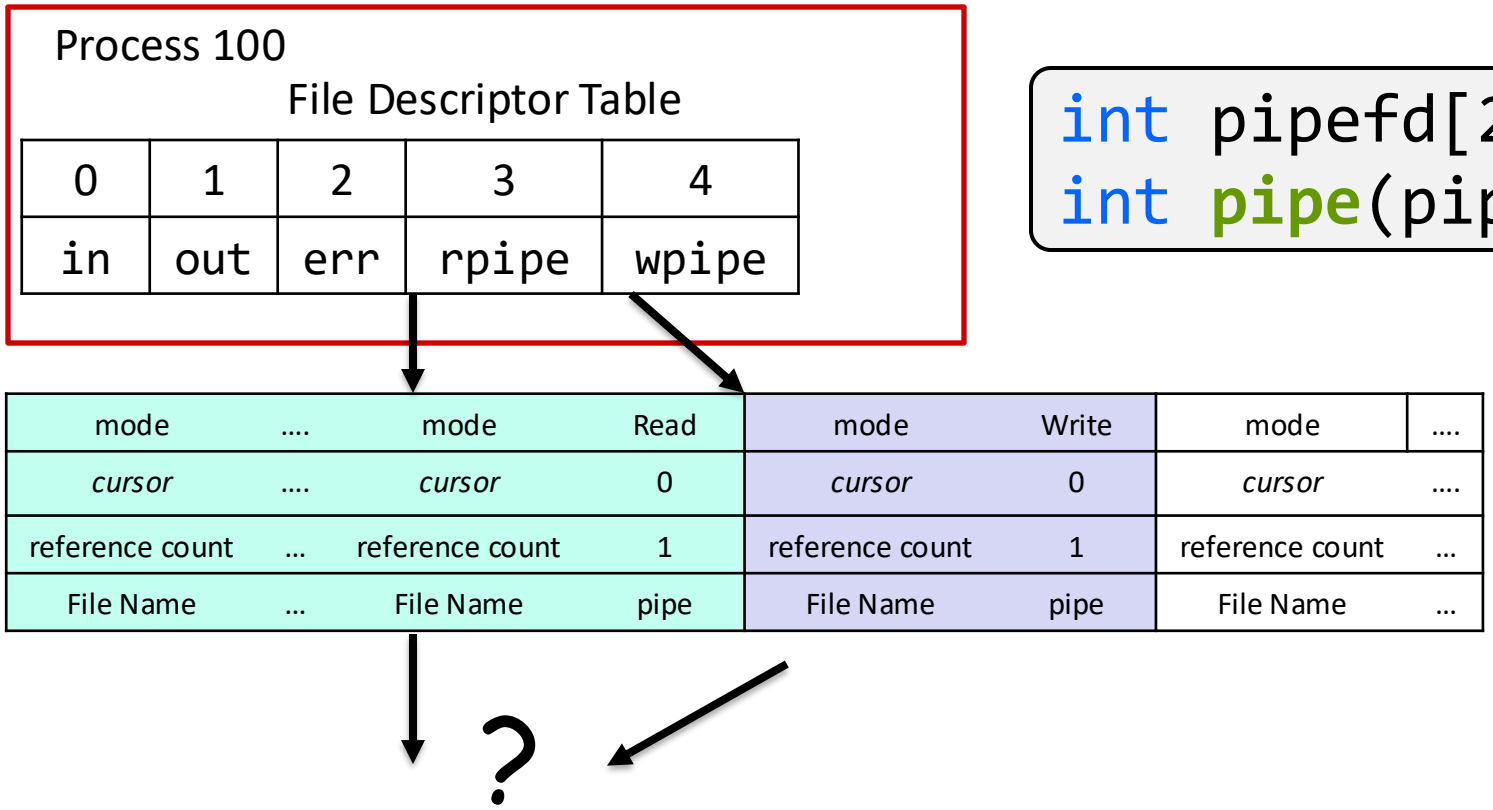
0	1	2
in	out	err

```
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

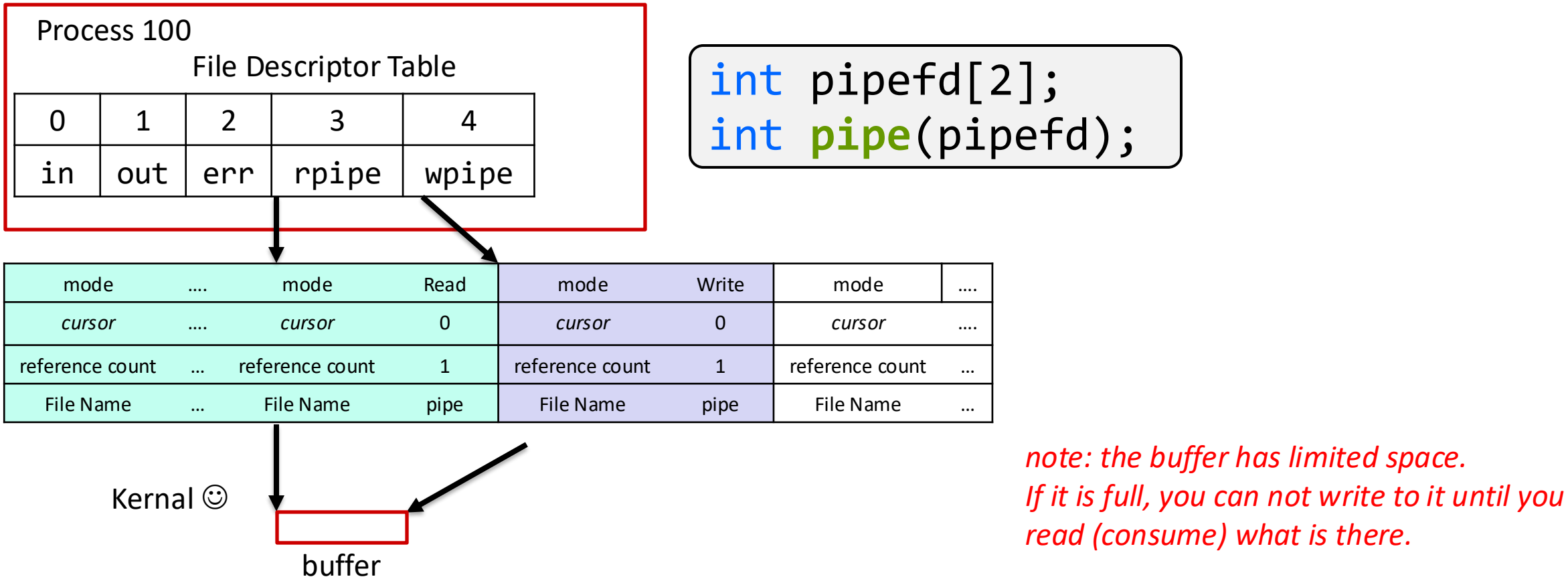
- Creating a pipe initializes two file descriptors in the process FD Table.
- Makes two entries in the system wide file table!




```
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!



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/cis5480

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

```c
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 side of the pipe is this?
        write(pipe_fds[1], message, strlen(message) + 1);

        char str[strlen(message) + 1];
        ssize_t chars_read = read(pipe_fds[1], message, sizeof(str));
        if(chars_read != -1){
            printf("%s", str);
        }
        return EXIT_SUCCESS;
    }

    close(pipe_fds[1]); //which side 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

pollev.com/cis5480

- 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;
}
```

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 it's 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

- Intro to file descriptors
- File Descriptors: Big Picture
- 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.

./..

- "/" is used to connect directory and file names together to create a file path.
  - E.g. "**`workspace/595/hello/`**"
- "." 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. If the file already exists, it just updates the file's time stamp

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 some contents. Prints out the line count, word count, and byte count
- "**`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-shredder < test_case"`

- `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/cis5480

- 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.

pollev.com/cis5480

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

pollev.com/cis5480

- 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?

Penn-Shell

- Making sense of all of this;
  - Forking, Signal Handlers, Masking, Exec*
  - File Descriptors, `open`/`close`/`read`/`write`, `dup2`, `pipe`…

Goodluck!