Pipe & File Descriptors (Continued)

Operating Systems, Fall 2026

Instructors: Travis McGaha

Head TAs: Shriya Sane

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

pollev.com/tqm

- Any questions/comments/concerns?

Administrivia

- Second Assignment (penn-shell)
  - Milestone 0 due Friday 9/11 // you have everything you need for this
  - Milestone 1 due Friday 9/18 // you will have everything you need after this lecture
  - Full assignment due Wednesday 9/23 // after Tuesday (9/15) lecture
  - Basic shell implementation
    - System calls, processes, I/O redirection, pipes, job control, and more!
  - Autograders are posted

Lecture Outline

- **Unix Commands & Controls**
- Review
  - File Descriptors
  - File Table
  - Open File Table
- Pipes and Dup
- pipe2

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.

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

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.

Discuss

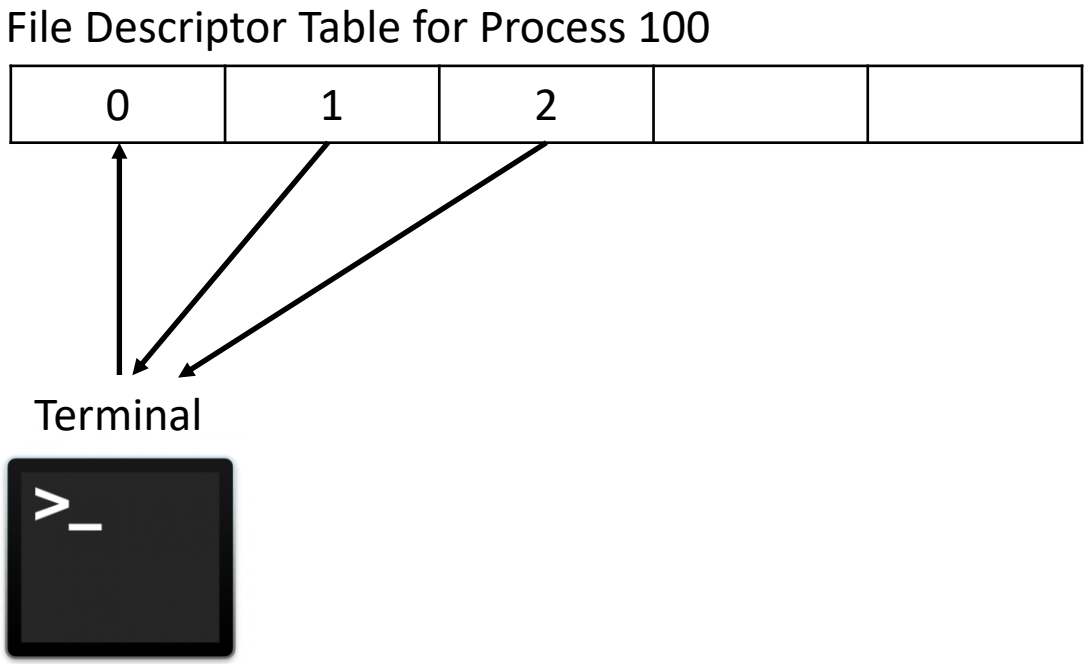
- If there's time, how would we even implement this?
  - `ls | sort -r`
- How many processes are necessary, do we need pipes, what about dup2?

Lecture Outline

- Unix Commands & Controls
- **Review**
  - **File Descriptors**
  - **File Table**
  - **Open File Table**
- Pipes and Dup
- pipe2

File Descriptor Table

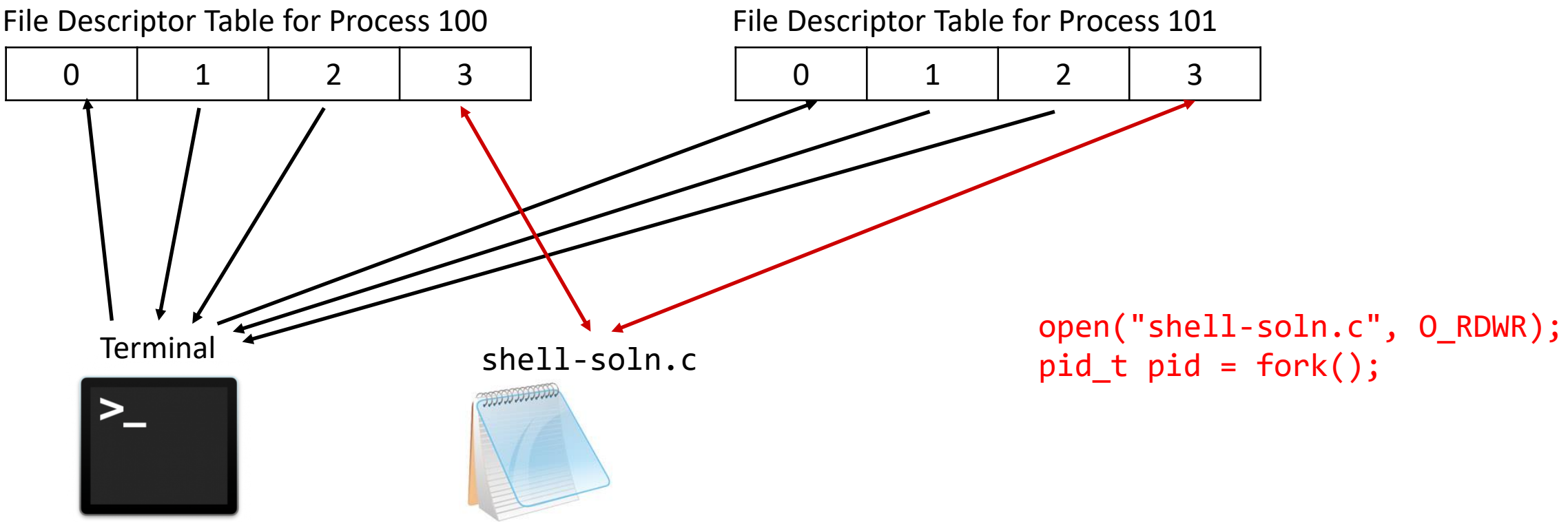
- Each process has **its own file descriptor table** managed by the OS
  - The table maintains information about the respective files the process has references to
- A *file descriptor* is an index into a processes FD table



Not an accurate depiction.

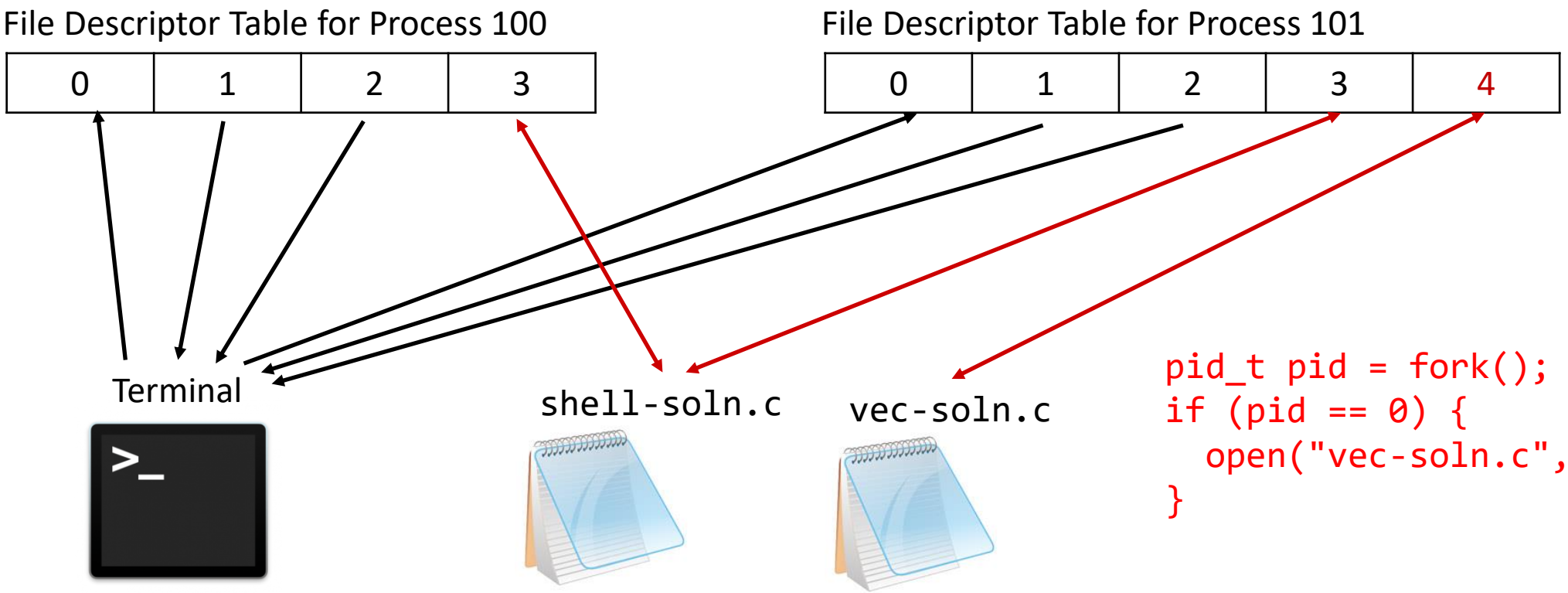
File Descriptor Table and Fork

- Fork will make ***an IDENTICAL copy of the parent's file descriptor table***
- If a file is opened *before* forking, child processes will inherit that file descriptor from the parent & point to the same file reference!



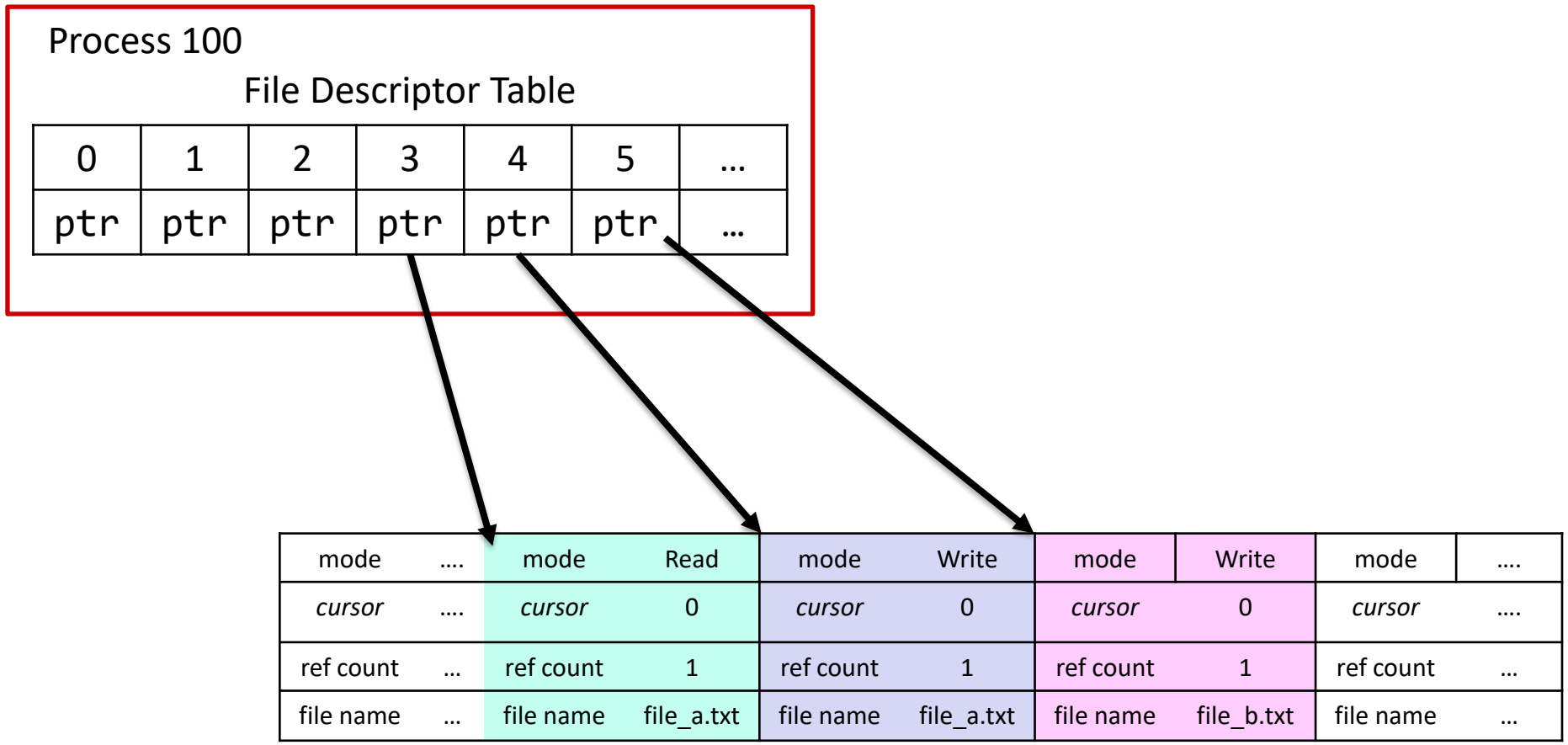
File Descriptor Table and Fork

- Fork will make ***an IDENTICAL copy of the parent's file descriptor table***
- If a file is opened *after* forking, the file reference *will not* be shared between the parent & child!



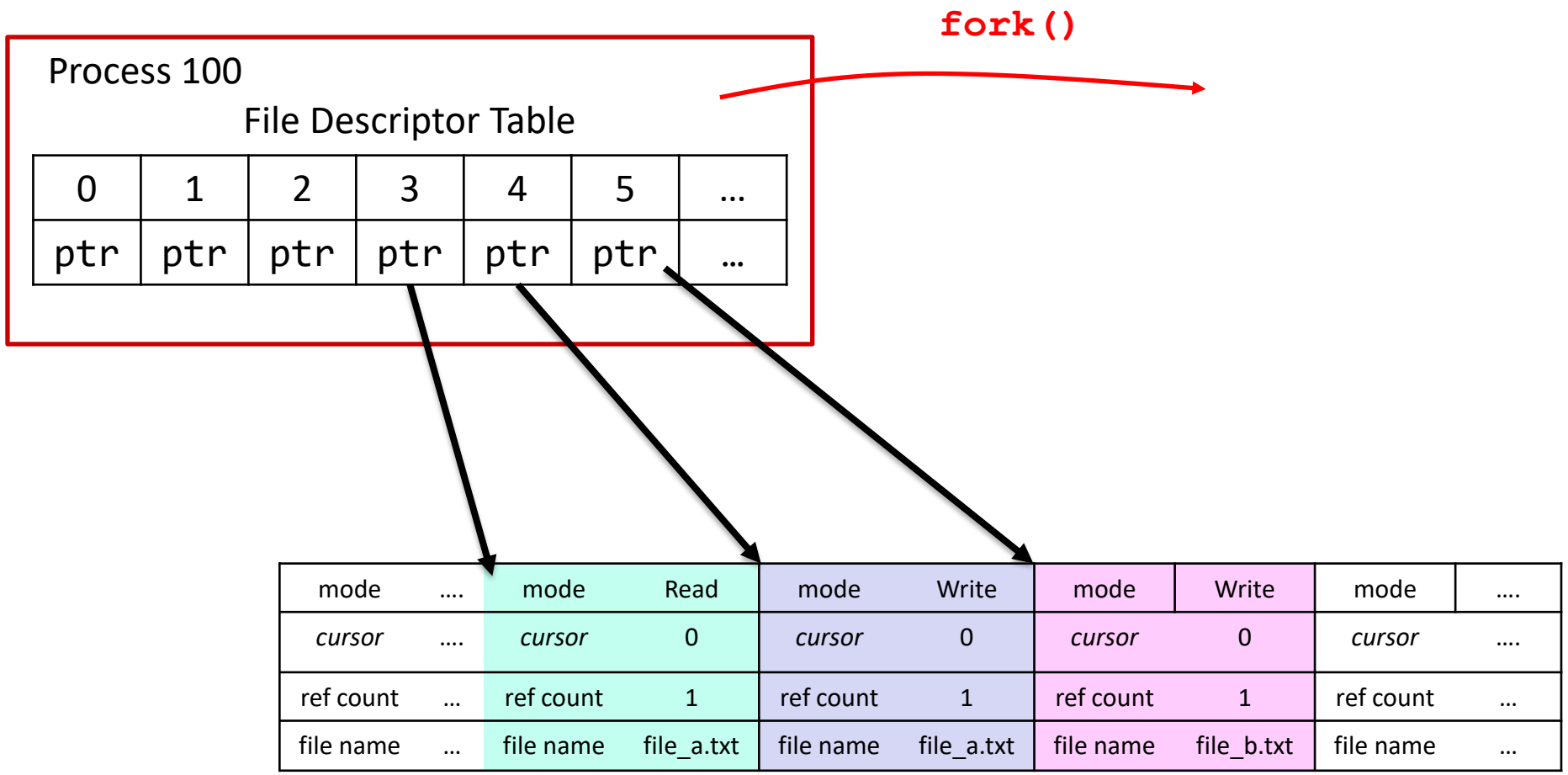

```
pid_t pid = fork();
if (pid == 0) {
  open("vec-soln.c", O_RDWR);
}
```

The Open File Table



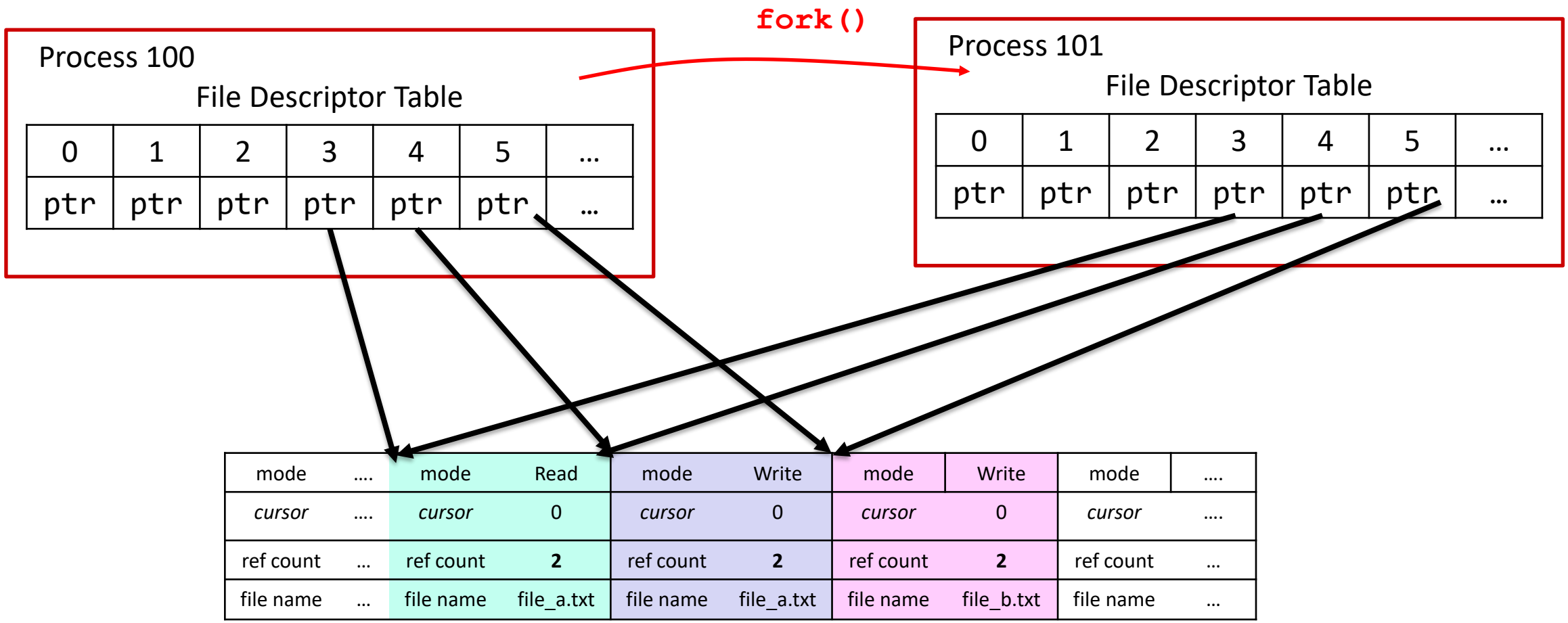
*The v/inode row is removed since it's not relevant at the moment.

The Open File Table



*The v/inode row is removed since it's not relevant at the moment.

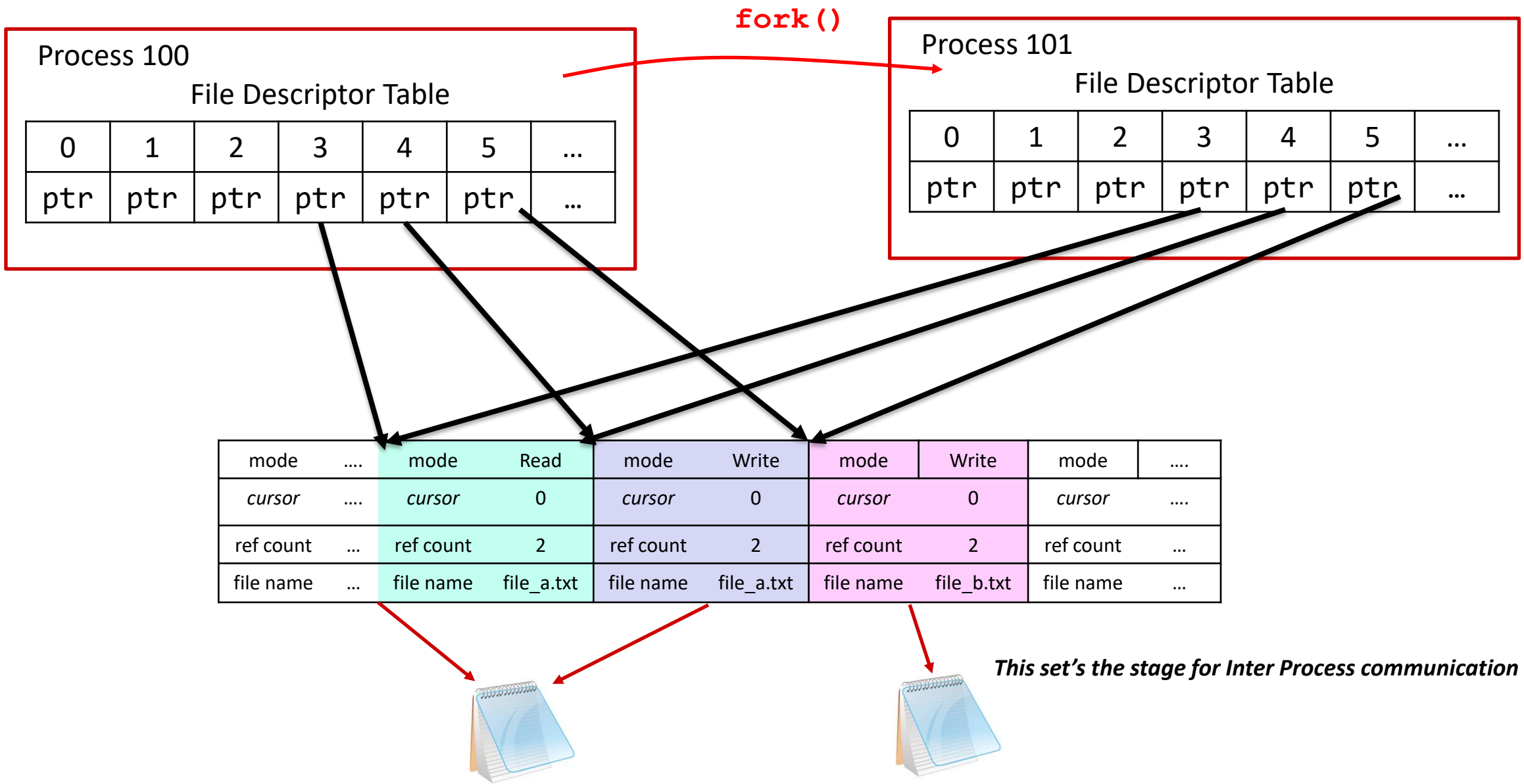
The Open File Table



Reference counts are incremented with fork!

*The v/inode row is removed since it's not relevant at the moment.

The Open File Table



This set's the stage for Inter Process communication via pipes…

File Descriptors and Exec

- Recall the family of `exec*` functions are C standard library functions layered on top of the `execve` system call

- From `man 2 execve`
  - *"By default, file descriptors remain open across an **execve**()."*

- What are the implications of this?
  - What if we redirect one of the standard FDs?
  - Do we always want this behavior?

Lecture Outline

- Unix Commands & Controls
- Review
  - File Descriptors
  - File Table
  - Open File Table
- **Pipes and Dup**
- pipe2

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

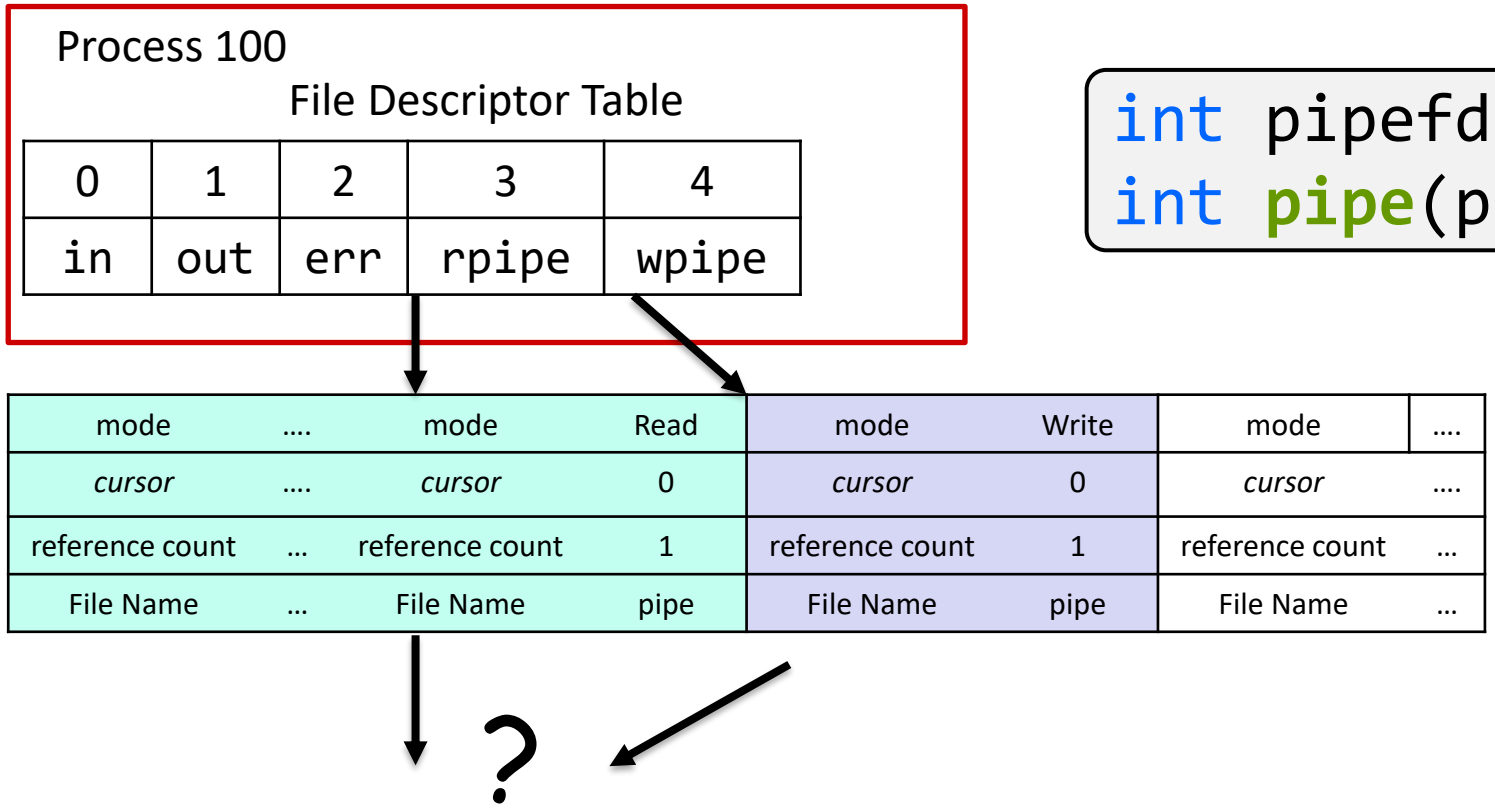
0	1	2
in	out	err

```
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

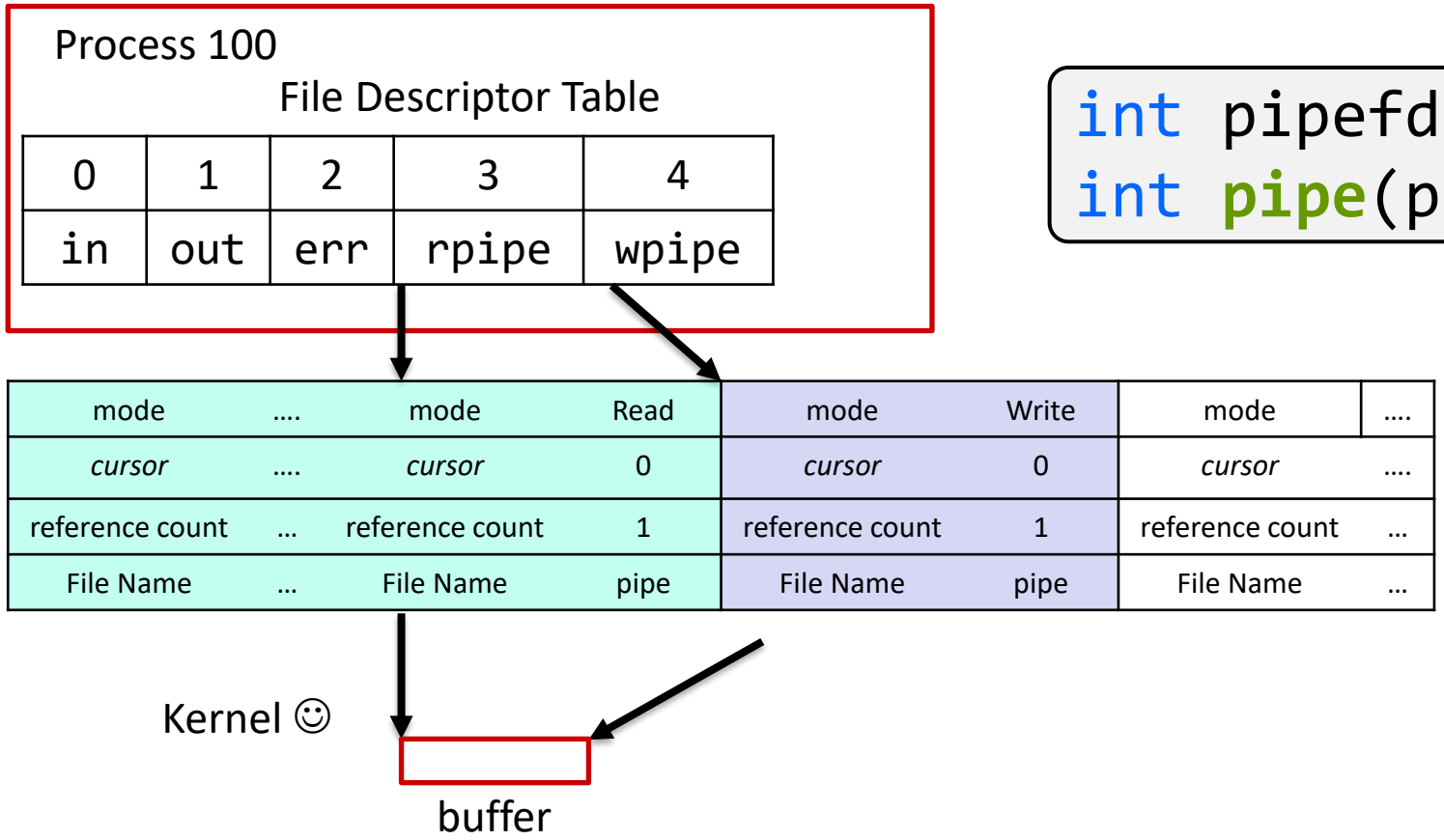
- Creating a pipe initializes two file descriptors in the process FD Table.
- Makes two entries in the system wide file table!




```
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.

Interprocess Communication: *Pipes*

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

- 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

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

- Pipe exists while at least *one* process has a reference to it, but unlike a file its data is *not* persistent!
  - Reading from the pipe consumes what was written

Poll Everywhere

pollev.com/tqm

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

```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 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

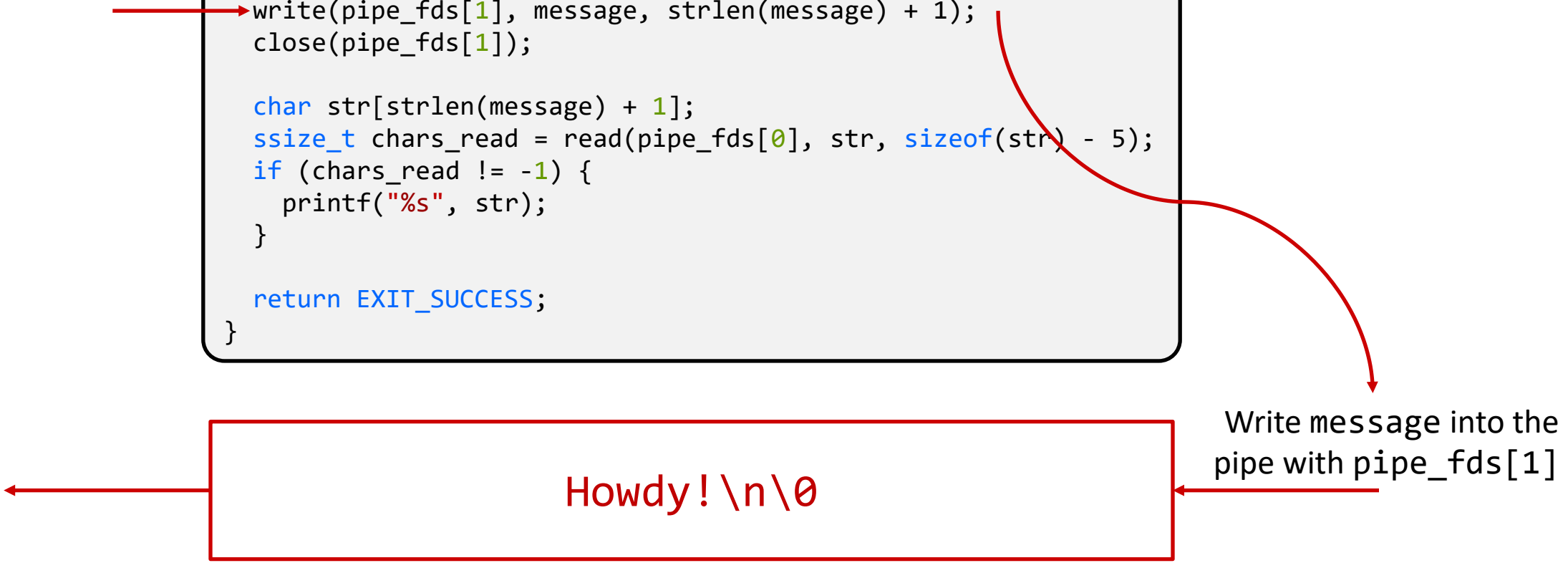
- When ***reading,*** you read until you've read enough bytes or EOF.
- ***When using a pipe, if there is nothing in the buffer to read, then you will wait until there is.***
  - ***EOF will not be returned nor will 0, when there is nothing in the pipes buffer.***
  - ***You must write to the write end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

Buffer Example: pipe_single_proc.c

```c
int main() {
  int pipe_fds[2];
  pipe(pipe_fds);

  char *message = "Howdy!\n";
  write(pipe_fds[1], message, strlen(message) + 1);
  close(pipe_fds[1]);

  char str[strlen(message) + 1];
  ssize_t chars_read = read(pipe_fds[0], str, sizeof(str) - 5);
  if (chars_read != -1) {
    printf("%s", str);
  }

  return EXIT_SUCCESS;
}
```



Buffer Example: pipe_single_proc.c

```c
int main() {
  int pipe_fds[2];
  pipe(pipe_fds);

  char *message = "Howdy!\n";
  write(pipe_fds[1], message, strlen(message) + 1);
  close(pipe_fds[1]);

  char str[strlen(message) + 1];
  ssize_t chars_read = read(pipe_fds[0], str, sizeof(str) - 5);
  if (chars_read != -1) {
    printf("%s", str);
  }

  return EXIT_SUCCESS;
}
```

Howdy!\n\0

Buffer Example: pipe_single_proc.c

```c
int main() {
  int pipe_fds[2];
  pipe(pipe_fds);

  char *message = "Howdy!\n";
  write(pipe_fds[1], message, strlen(message) + 1);
  close(pipe_fds[1]);

  char str[strlen(message) + 1];
  ssize_t chars_read = read(pipe_fds[0], str, sizeof(str) - 5);
  if (chars_read != -1) {
    printf("%s", str);
  }

  return EXIT_SUCCESS;
}
```

Read 3 bytes from the pipe with `pipe_fds[0]`

Howdy!\n\0

Buffer Example: pipe_single_proc.c

```c
int main() {
  int pipe_fds[2];
  pipe(pipe_fds);

  char *message = "Howdy!\n";
  write(pipe_fds[1], message, strlen(message) + 1);
  close(pipe_fds[1]);

  char str[strlen(message) + 1];
  ssize_t chars_read = read(pipe_fds[0], str, sizeof(str) - 5);
  if (chars_read != -1) {
    printf("%s", str);
  }

  return EXIT_SUCCESS;
}
```

How

Read 3 bytes from the pipe with `pipe_fds[0]`

dy!\n\0

Buffer Example: pipe_single_proc.c

```c
int main() {
  int pipe_fds[2];
  pipe(pipe_fds);

  char *message = "Howdy!\n";
  write(pipe_fds[1], message, strlen(message) + 1);
  close(pipe_fds[1]);

  char str[strlen(message) + 1];
  ssize_t chars_read = read(pipe_fds[0], str, sizeof(str) - 5);
  if (chars_read != -1) {
    printf("%s", str);
  }

  return EXIT_SUCCESS;
}
```

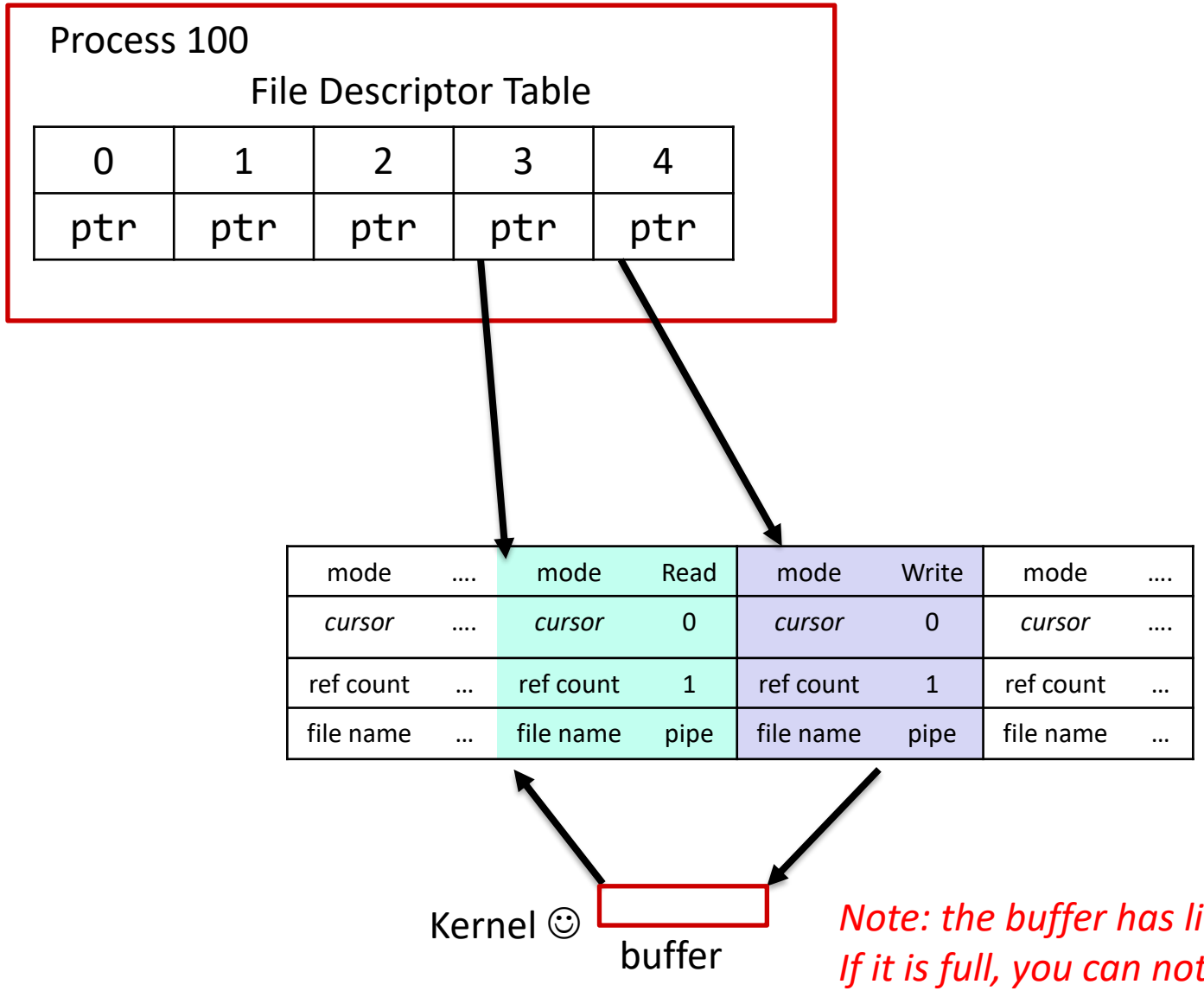
Print: How →

← dy!\n\0

Buffer Example: pipe_single_proc.c

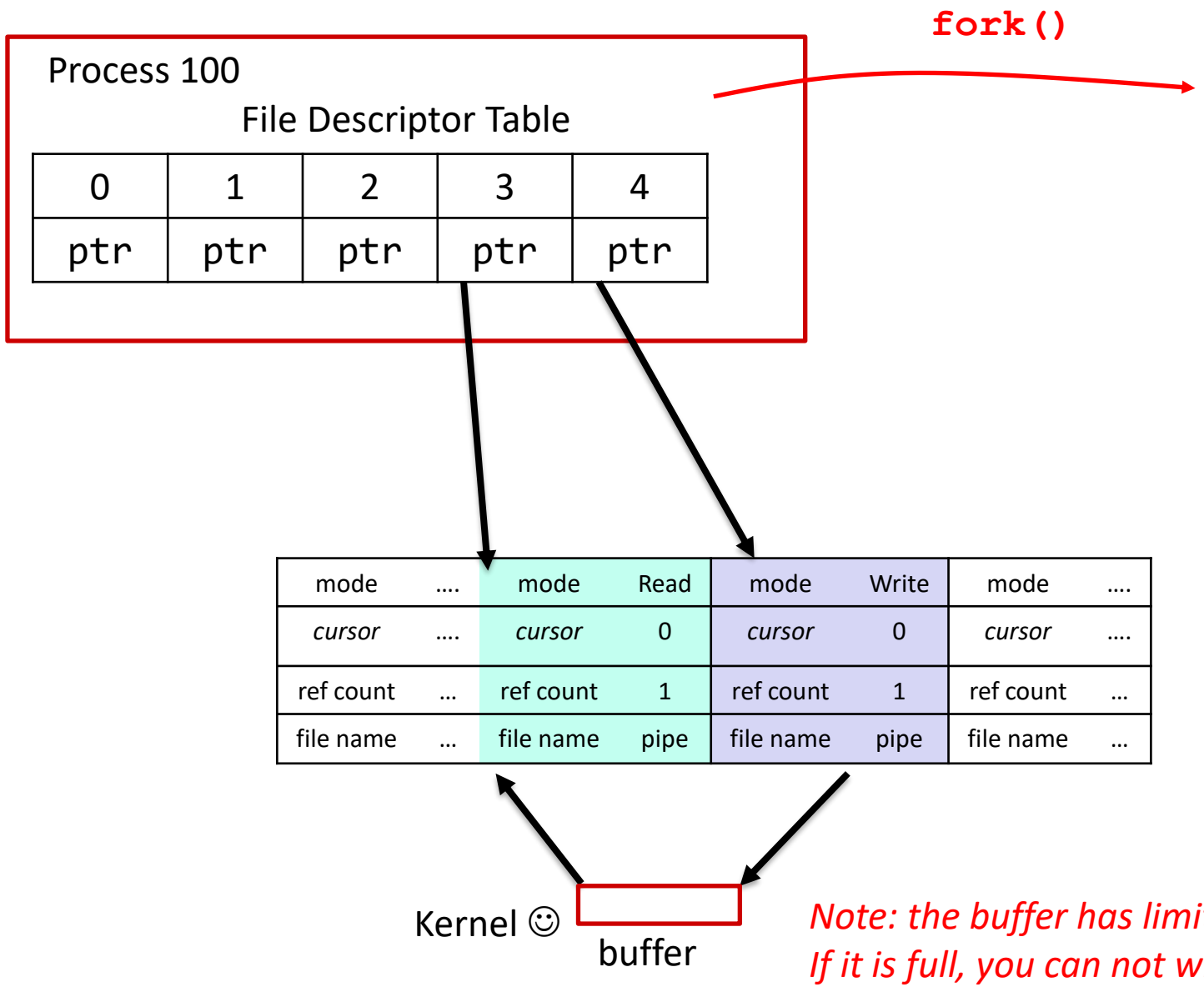
```c
int main() {
  int pipe_fds[2];
  pipe(pipe_fds);

  char *message = "Howdy!\n";
  write(pipe_fds[1], message, strlen(message) + 1);
  close(pipe_fds[1]);

  char str[strlen(message) + 1];
  ssize_t chars_read = read(pipe_fds[0], str, sizeof(str) - 5);
  if (chars_read != -1) {
    printf("%s", str);
  }

  return EXIT_SUCCESS;
}
```

Visualizing Pipes with Fork



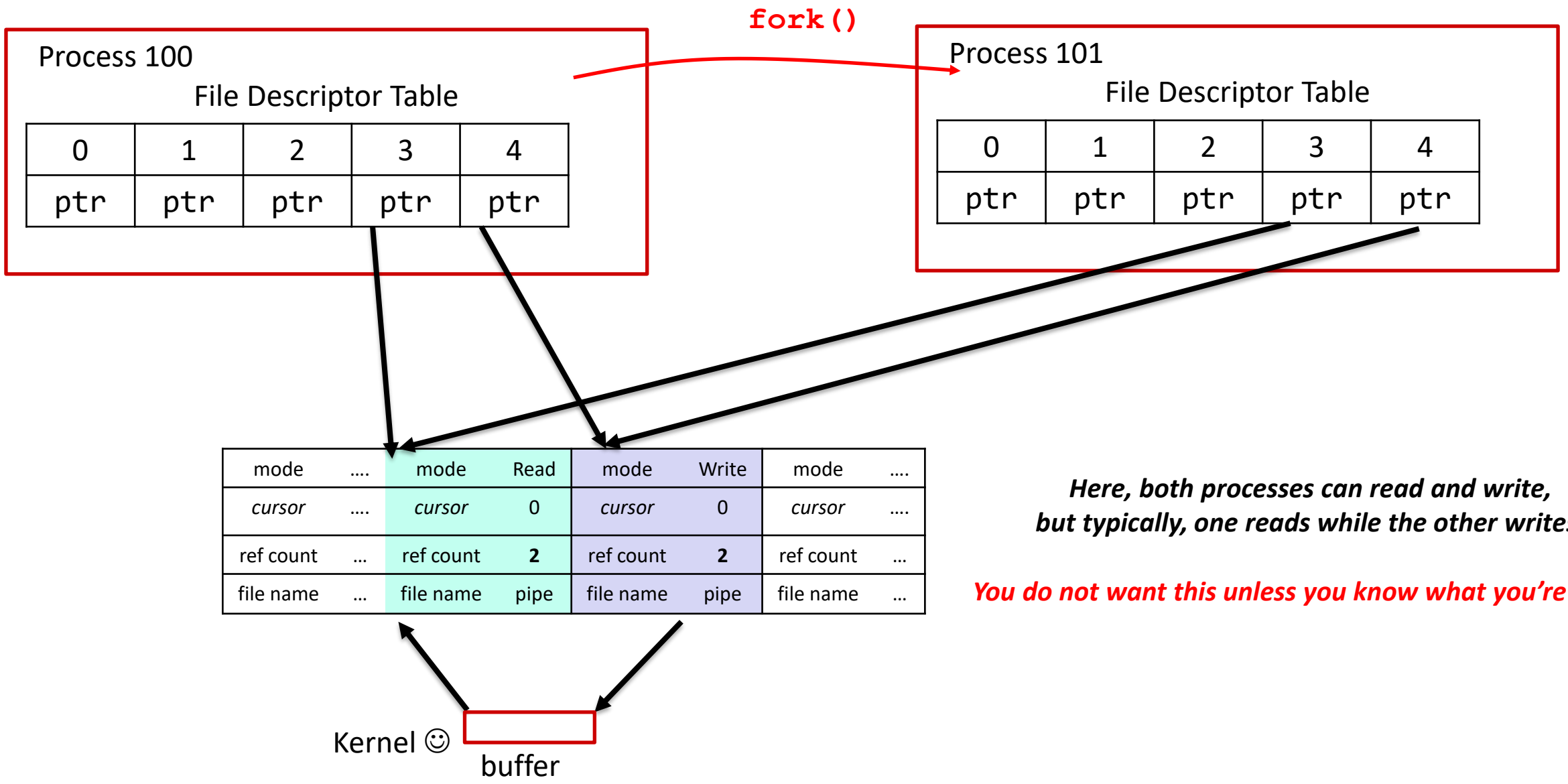
Note: the buffer has limited space.
If it is full, you can not write to it until you read (consume) what is there.

Visualizing Pipes with Fork



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

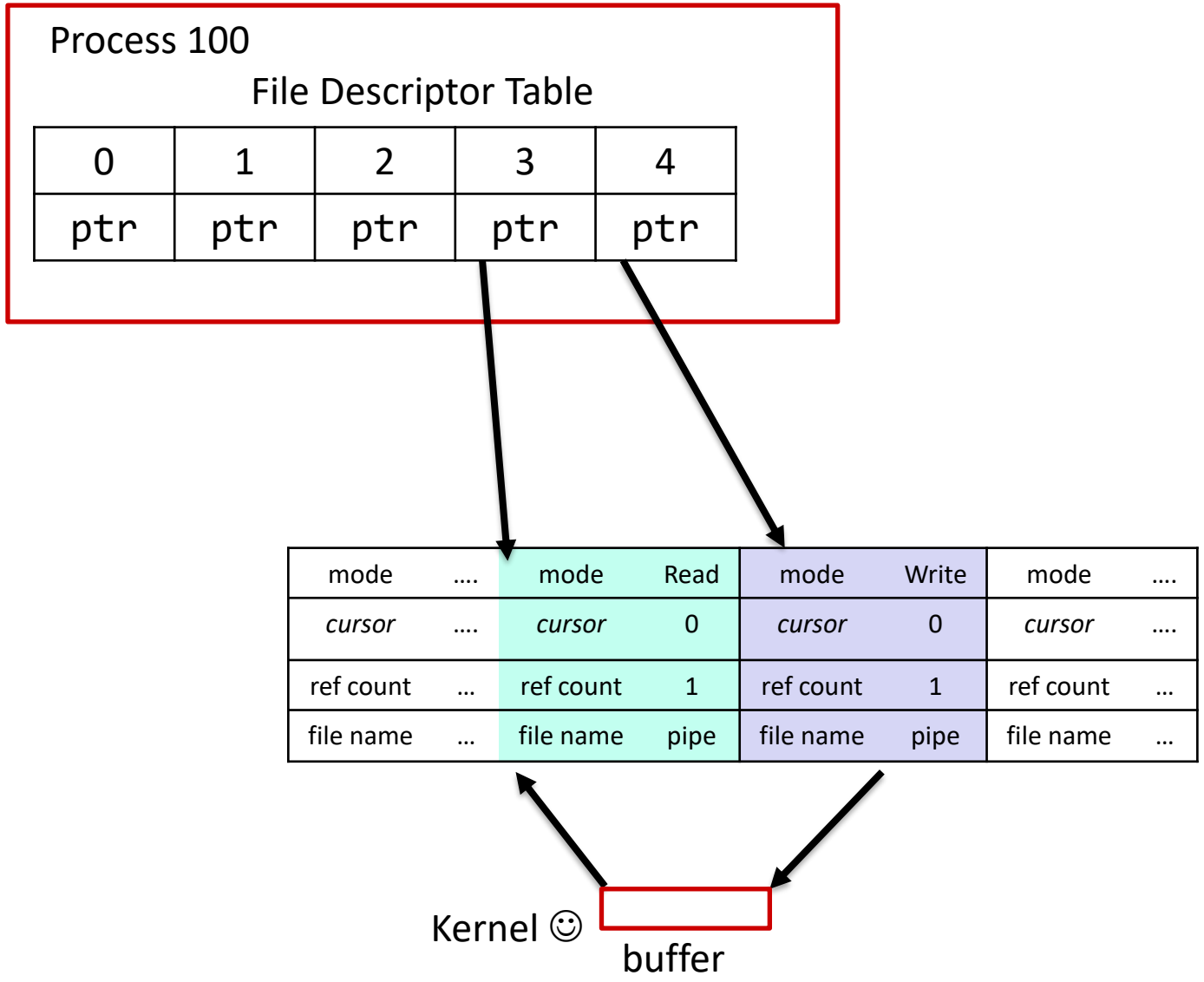
Visualizing Pipes with Fork



Here, both processes can read and write, but typically, one reads while the other writes.

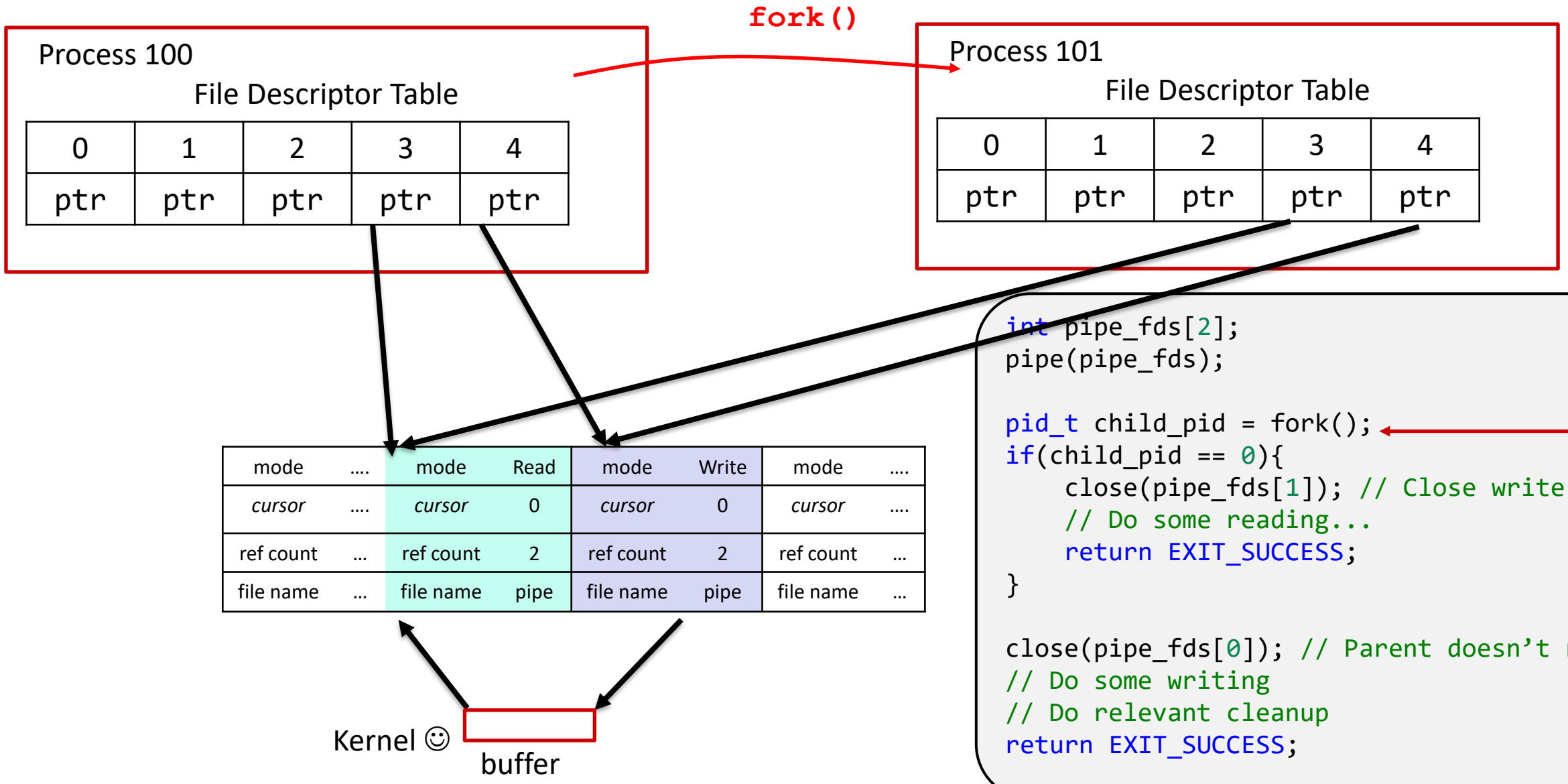
You do not want this unless you know what you're doing.

Short Program Walkthrough: `basic_pipe.c`



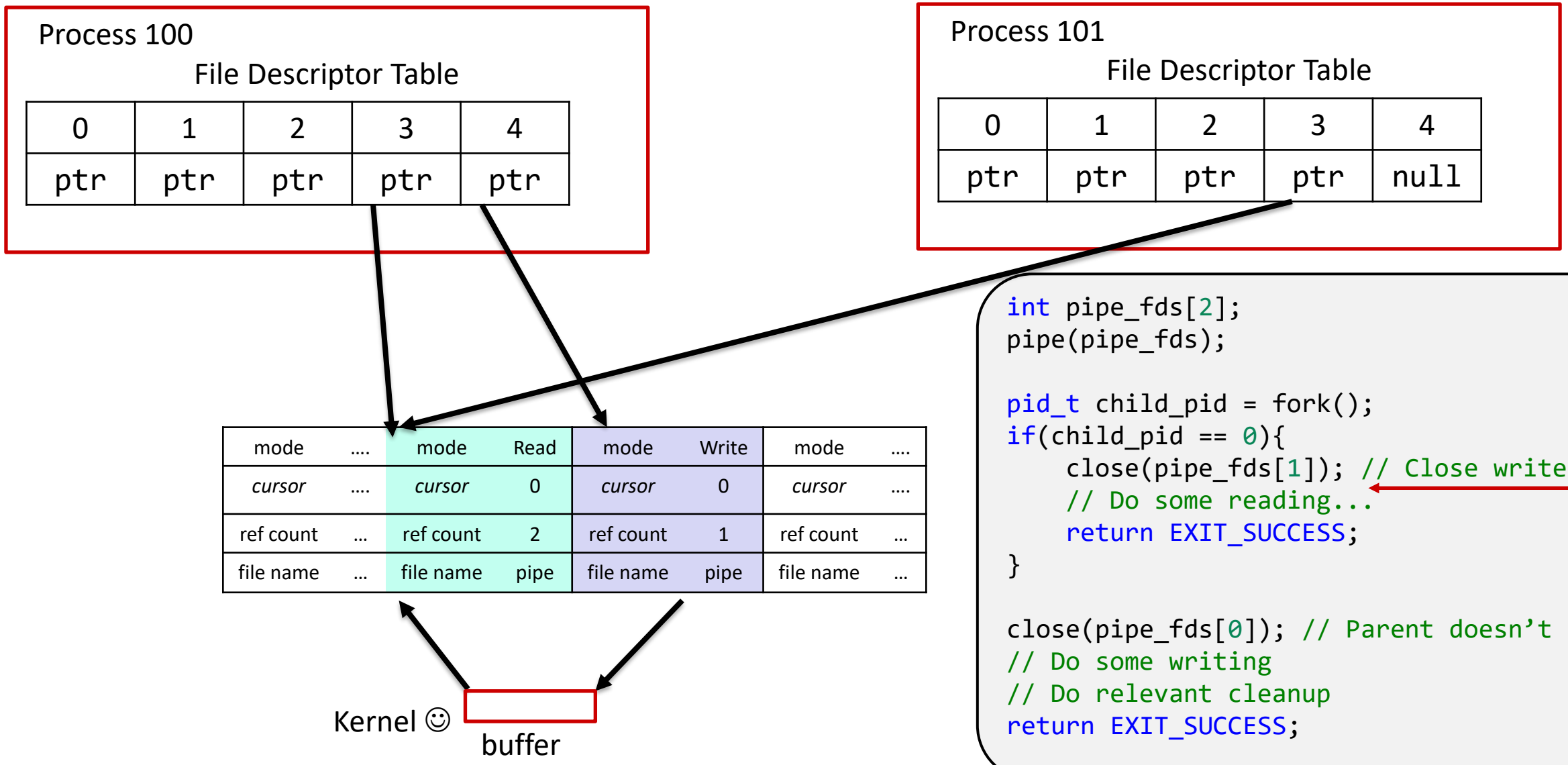

```c
int pipe_fds[2];
pipe(pipe_fds);

pid_t child_pid = fork();
if(child_pid == 0){
    close(pipe_fds[1]); // Close write end
    // Do some reading...
    return EXIT_SUCCESS;
}

close(pipe_fds[0]); // Parent doesn't read
// Do some writing
// Do relevant cleanup
return EXIT_SUCCESS;
```

Short Program Walkthrough: `basic_pipe.c`




```c
int pipe_fds[2];
pipe(pipe_fds);

pid_t child_pid = fork();
if(child_pid == 0){
    close(pipe_fds[1]); // Close write end
    // Do some reading...
    return EXIT_SUCCESS;
}

close(pipe_fds[0]); // Parent doesn't read
// Do some writing
// Do relevant cleanup
return EXIT_SUCCESS;
```

Short Program Walkthrough: `basic_pipe.c`




```c
int pipe_fds[2];
pipe(pipe_fds);

pid_t child_pid = fork();
if(child_pid == 0){
    close(pipe_fds[1]); // Close write end
    // Do some reading...
    return EXIT_SUCCESS;
}

close(pipe_fds[0]); // Parent doesn't read
// Do some writing
// Do relevant cleanup
return EXIT_SUCCESS;
```

Short Program Walkthrough: `basic_pipe.c`

Process 100

File Descriptor Table

0	1	2	3	4
ptr	ptr	ptr	null	ptr

Process 101

File Descriptor Table

0	1	2	3	4
ptr	ptr	ptr	ptr	null

mode	....	mode	Read	mode	Write	mode	....
cursor	....	*cursor*	0	*cursor*	0	*cursor*	....
ref count	...	ref count	1	ref count	1	ref count	...
file name	...	file name	pipe	file name	pipe	file name	...

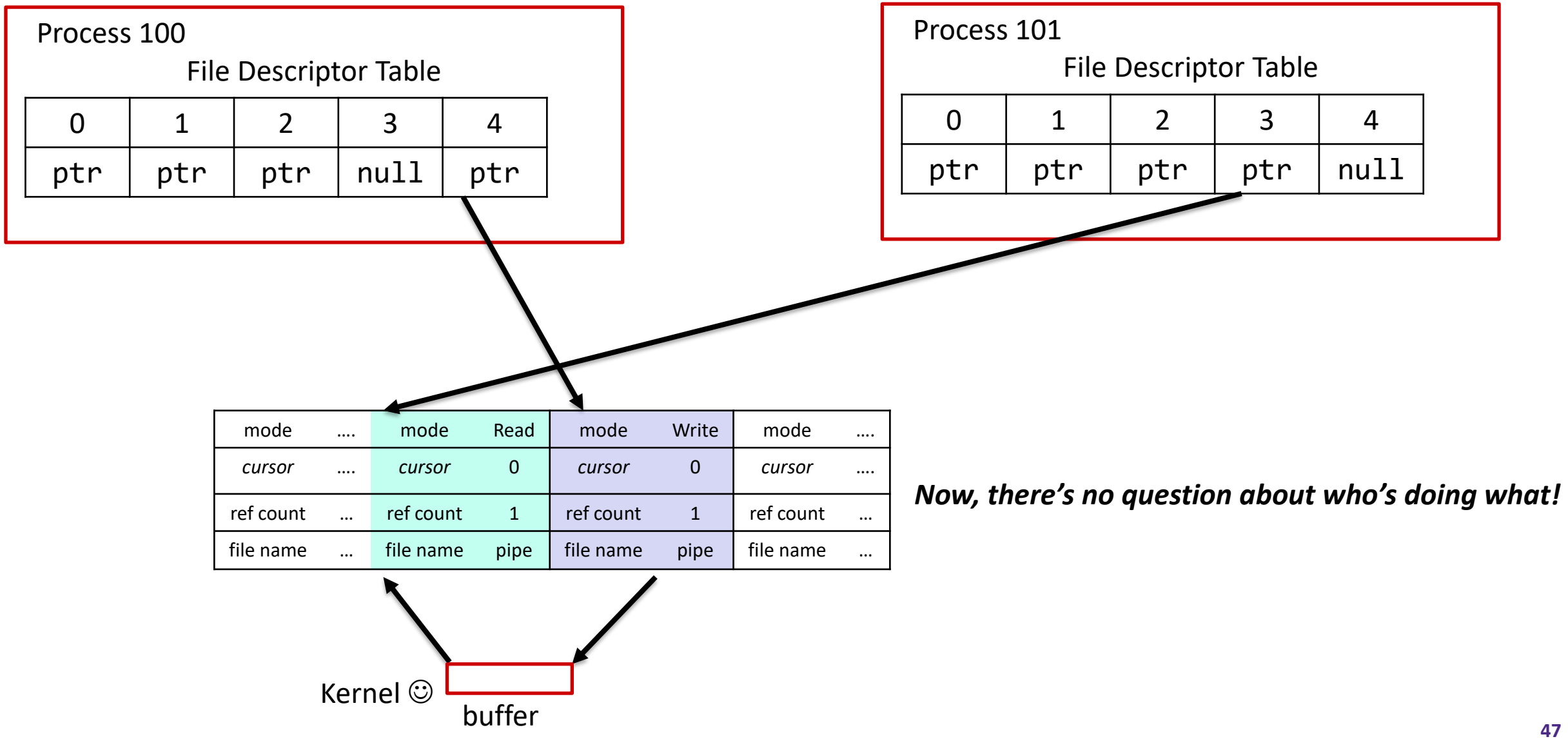
Kernel ☺ buffer

```c
int pipe_fds[2];
pipe(pipe_fds);

pid_t child_pid = fork();
if(child_pid == 0){
    close(pipe_fds[1]); // Close write end
    // Do some reading...
    return EXIT_SUCCESS;
}

close(pipe_fds[0]); // Parent doesn't read
// Do some writing
// Do relevant cleanup
return EXIT_SUCCESS;
```

Short Program Final State



Now, there's no question about who's doing what!

`pipe_single_proc.c`: Revisited

- When reading, you read until you've read the requested number of bytes or EOF.
  - What if we did not close the write end of the pipe and tried to read again from an empty pipe?

- For files, reaching the end will return an End-of-File (EOF) character, but for pipes an empty pipe *does not mean* the pipe is done being used!

- ***EOF is only read from a pipe when the pipe is empty and all write ends of the pipe are closed so it's impossible for any new data to show up!***

File Redirection with dup2

- 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***
```

Unix Shell Control Operators

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

- **`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`**

Piping in the Shell

```
cat manual.txt | grep Act | uniq
```

- **`cat`** first outputs the entire contents of `manual.txt` and pipes it into **`grep`**, which filters for lines containing `"Act"`
- The output from **`grep`** is then piped into the **`uniq`** command, which removes duplicate lines from the output, ensuring each matching line appears only once
- What would the fd table (for each process) and open file need to look like to make this feasible?

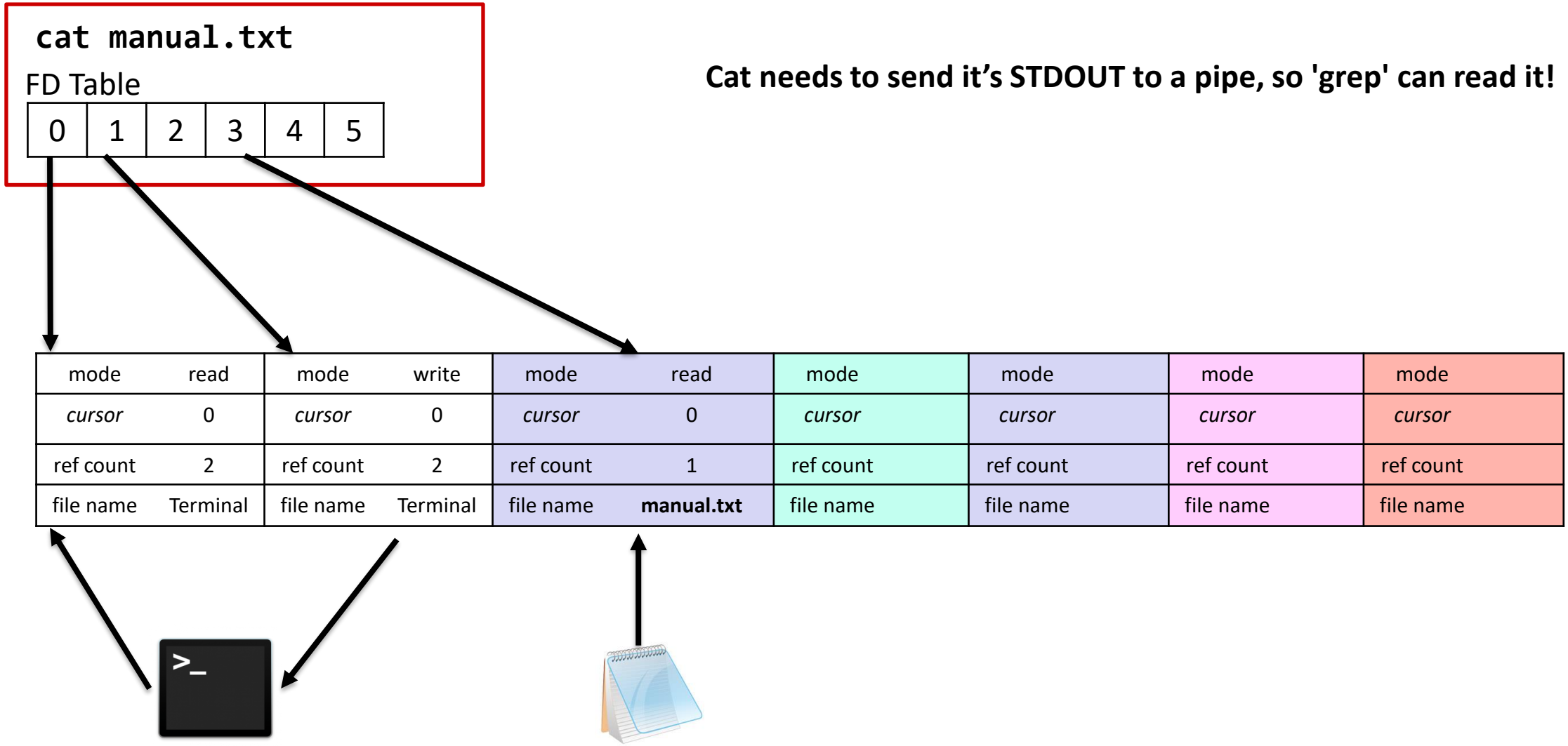
Important: it is the shell process that forks each of these processes and intertwines their pipes together.

Raise Fingers

How many pipes do we need to execute this command?

```
cat manual.txt | grep Act | uniq
```

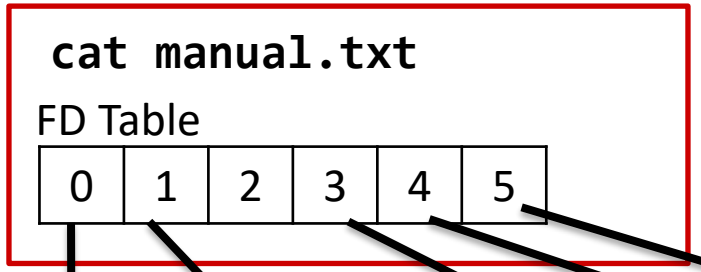
cat manual.txt | grep Act | uniq



Cat needs to send it's STDOUT to a pipe, so 'grep' can read it!

Note: the ref counts might seem inflated, but there is a shell process that exists too and forks these processes.

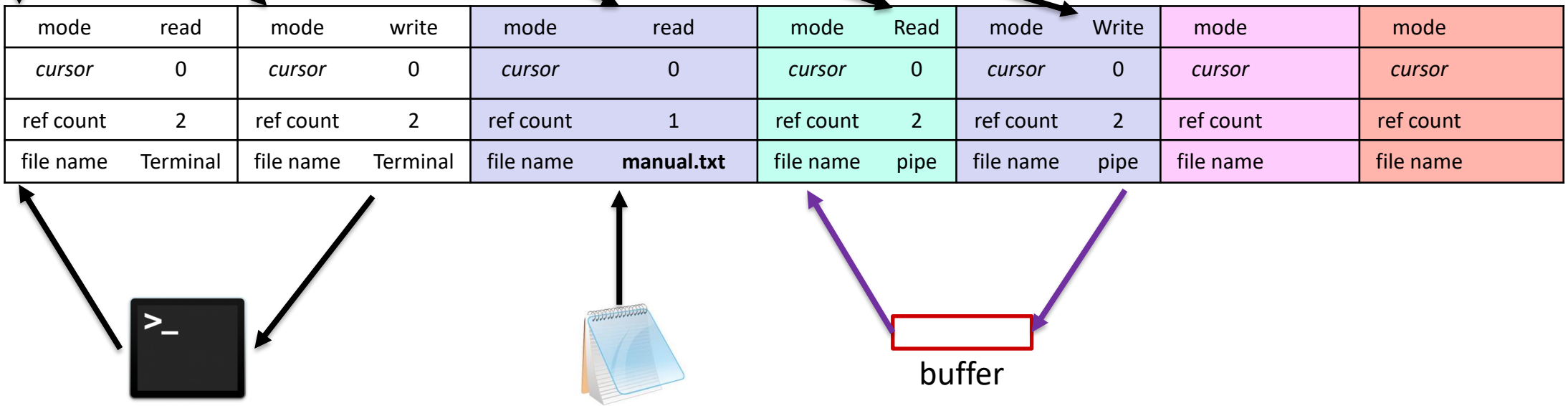
cat manual.txt | grep Act | uniq



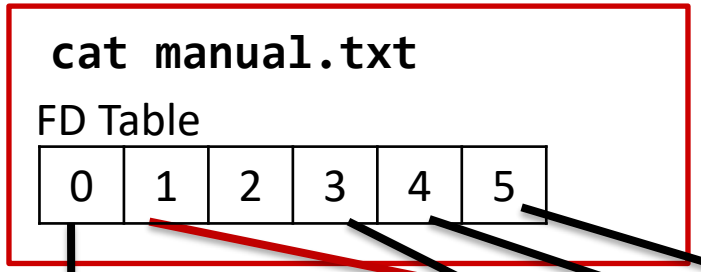
Cat needs to send it's STDOUT to a pipe, so 'grep' can read it!

1. We need to make a pipe, via ***pipe()***
2. We need to dup2 with STDOUT and the ***WRITE*** portion of the pipe…

```
dup2(cat_pipe[1], STDOUT_FILENO);
```



cat manual.txt | grep Act | uniq

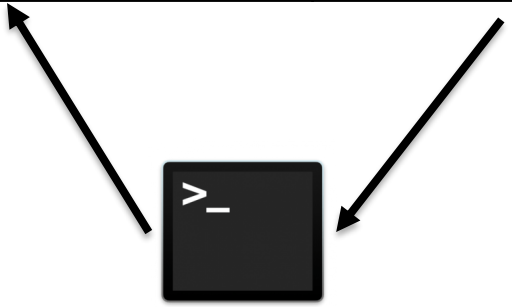


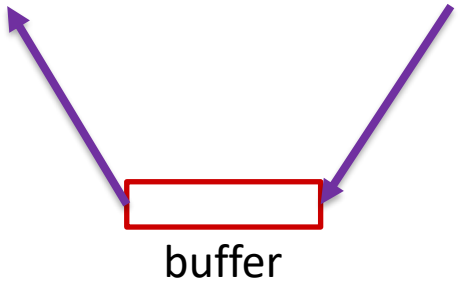
Cat needs to send it's STDOUT to a pipe, so 'grep' can read it!

1. We need to make a pipe, via ***pipe()***
2. We need to dup2 with STDOUT and the ***WRITE*** portion of the pipe before we exec!

```
dup2(cat_pipe[1], STDOUT_FILENO);
```

mode	read	mode	write	mode	read	mode	Read	mode	Write	mode	mode
cursor	0	*cursor*	0	*cursor*	0	*cursor*	0	*cursor*	0	*cursor*	*cursor*
ref count	2	ref count	1	ref count	1	ref count	2	ref count	3	ref count	ref count
file name	Terminal	file name	Terminal	file name	**manual.txt**	file name	pipe	file name	pipe	file name	file name





cat manual.txt | grep Act | uniq

cat manual.txt

FD Table

0	1	2	3	4	5

That's a lot of references to this pipe… do we need them all?
After dup2 cat doesn't need the read or write end of the pipe so we should close them!

```
close(cat_pipe[0]);
close(cat_pipe[1]);
```

mode	read	mode	write	mode	read	mode	Read	mode	Write	mode	mode
cursor	0	*cursor*	0	*cursor*	0	*cursor*	0	*cursor*	0	*cursor*	*cursor*
ref count	2	ref count	1	ref count	1	ref count	1	ref count	2	ref count	ref count
file name	Terminal	file name	Terminal	file name	**manual.txt**	file name	pipe	file name	pipe	file name	file name

buffer

Poll Everywhere

pollev.com/tqm

```
cat manual.txt | grep Act | uniq
```

Where can we put a pipe so both cat and grep can write and read, respectively?

```
int cat_pipe[2];

pipe(&cat_pipe); // A ←

pid_t cat_pid = fork();
pipe(&cat_pipe); // B ←
if(cat_pid == 0){
   // do cat stuff
   // maybe do some pipe stuff?
}
pipe(&cat_pipe); // C ←
pid_t grep_pid = fork();
pipe(&cat_pipe); // D ←
if(grep_pid == 0){
   // do grep stuff
   // maybe do some pipe stuff?
}
```

Poll Everywhere

pollev.com/tqm

```
cat manual.txt | grep Act | uniq
```

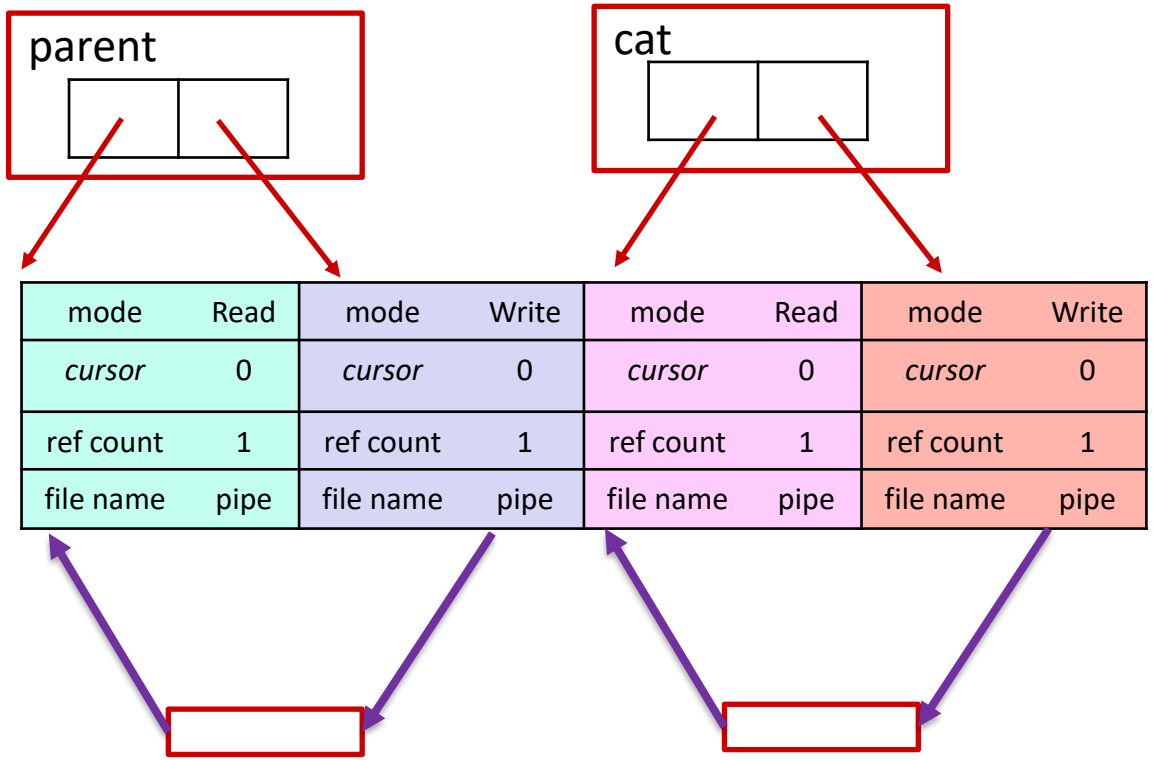
Where can we put a pipe so both cat and grep can write and read, respectively?

```
int cat_pipe[2];

pipe(&cat_pipe); // A ←

pid_t cat_pid = fork();
pipe(&cat_pipe); // B ←
if(cat_pid == 0){
   // do cat stuff
   // maybe do some pipe stuff?
}
pipe(&cat_pipe); // C ←
pid_t grep_pid = fork();
pipe(&cat_pipe); // D ←
if(grep_pid == 0){
   // do grep stuff
   // maybe do some pipe stuff?
}
```

B: ***If we pipe here, we make two sperate pipes, one in the parent process, and one in the cat process, this does not allow for cat and grep to share a pipe: why? The FDs are NOT SHARED!***



Poll Everywhere

pollev.com/tqm

```
cat manual.txt | grep Act | uniq
```

Where can we put a pipe so both cat and grep can write and read, respectively?

```
int cat_pipe[2];

pipe(&cat_pipe); // A ←

pid_t cat_pid = fork();
pipe(&cat_pipe); // B ←
if(cat_pid == 0){
   // do cat stuff
   // maybe do some pipe stuff?
}
pipe(&cat_pipe); // C ←
pid_t grep_pid = fork();
pipe(&cat_pipe); // D ←
if(grep_pid == 0){
   // do grep stuff
   // maybe do some pipe stuff?
}
```

C: ***If we pipe here, we make only one pipe, in the parent! The cat process has already gone off on its own. However, the grep process will inherit this pipe, just not the cat process.***

Recall: *"In Cat,* We need to dup2 with STDOUT and the ***WRITE*** portion of the pipe!"

How can we dup2 a pipe that never existed in the child process?

Poll Everywhere

pollev.com/tqm

```
cat manual.txt | grep Act | uniq
```

Where can we put a pipe so both cat and grep can write and read, respectively?

```
int cat_pipe[2];

pipe(&cat_pipe); // A ←

pid_t cat_pid = fork();
pipe(&cat_pipe); // B ←
if(cat_pid == 0){
   // do cat stuff
   // maybe do some pipe stuff?
}
pipe(&cat_pipe); // C ←
pid_t grep_pid = fork();
pipe(&cat_pipe); // D ←
if(grep_pid == 0){
   // do grep stuff
   // maybe do some pipe stuff?
}
```

D: ***This is similar to B, where we create a separate pipe in the parent and the grep process. No way to wrangle the pipes this way.***

Poll Everywhere

pollev.com/tqm

`cat manual.txt | grep Act | uniq`

Where can we put a pipe so both cat and grep can write and read, respectively?

```
int cat_pipe[2];

pipe(&cat_pipe); // A ←

pid_t cat_pid = fork();

if(cat_pid == 0){
   // do cat stuff
   // maybe do some pipe stuff?
}

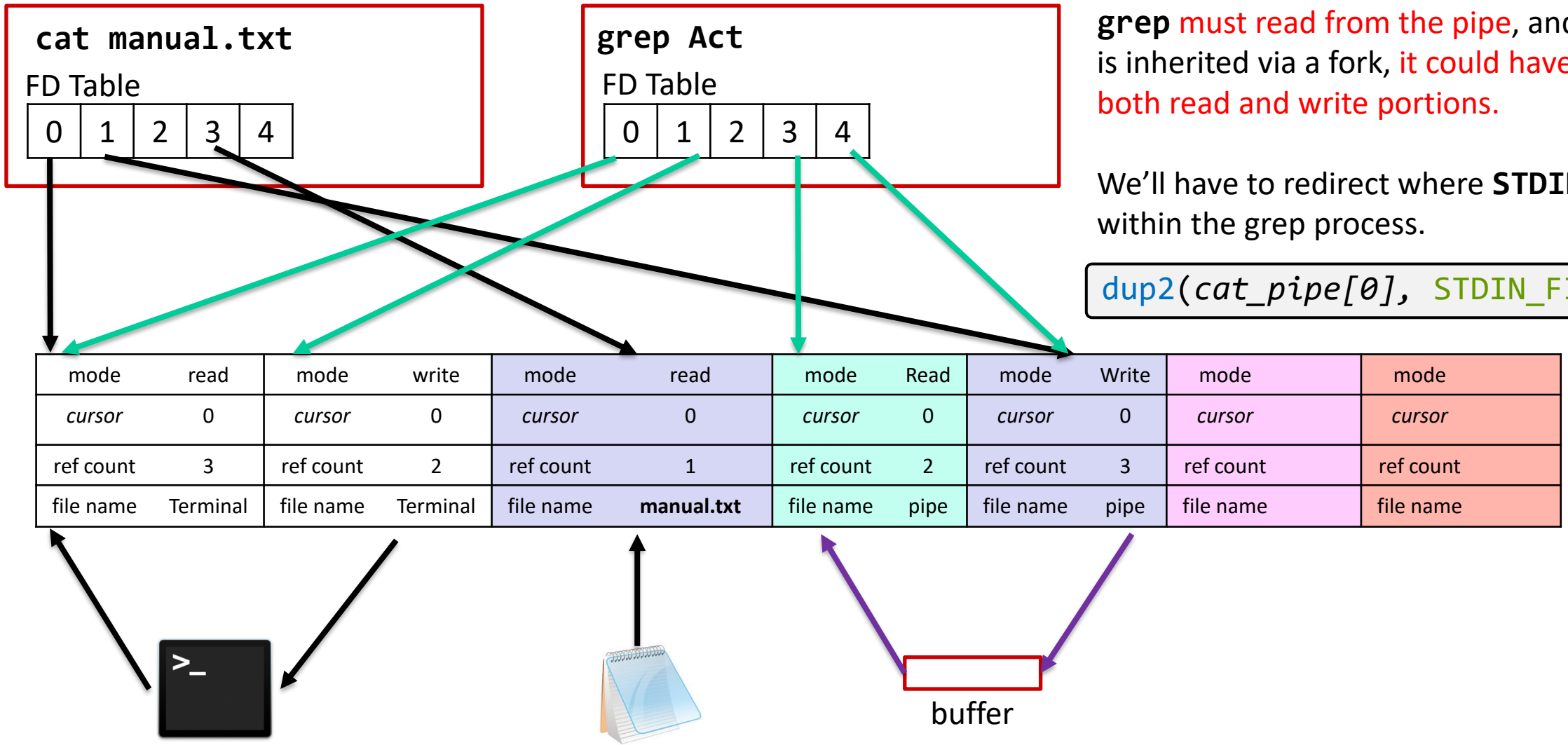
pid_t grep_pid = fork();

if(grep_pid == 0){
   // do grep stuff
   // maybe do some pipe stuff?
}
```

The answer is A:

The pipe must be created in the parent before forking **`cat`**. This way the pipe is in the parent and can be copied into the second child **`grep`**, which will need read access.

`cat manual.txt | grep Act | uniq`

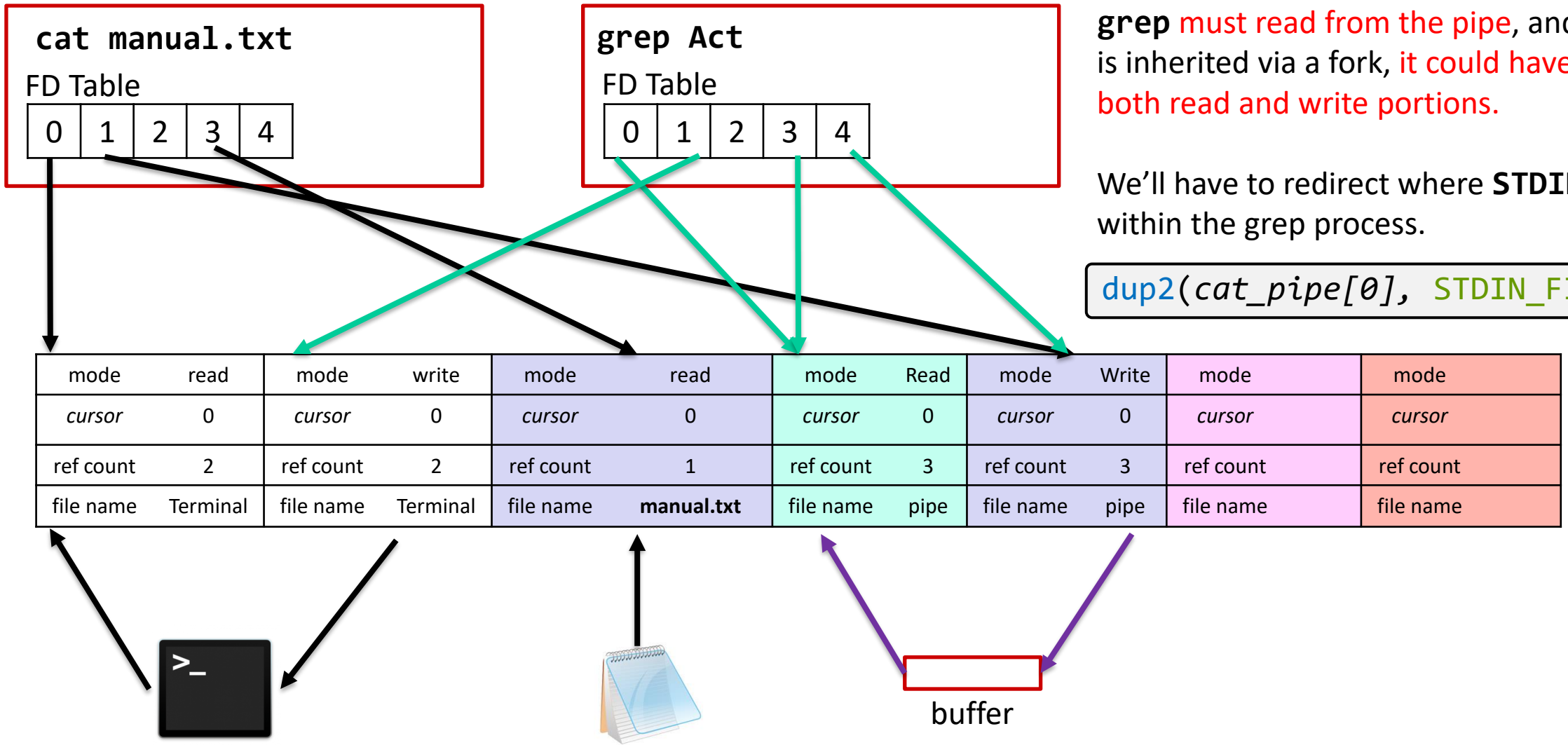


`grep` must read from the pipe, and as the pipe is inherited via a fork, it could have access to both read and write portions.

We'll have to redirect where **`STDIN`** refers to within the grep process.

```
dup2(cat_pipe[0], STDIN_FILENO);
```

cat manual.txt | grep Act | uniq

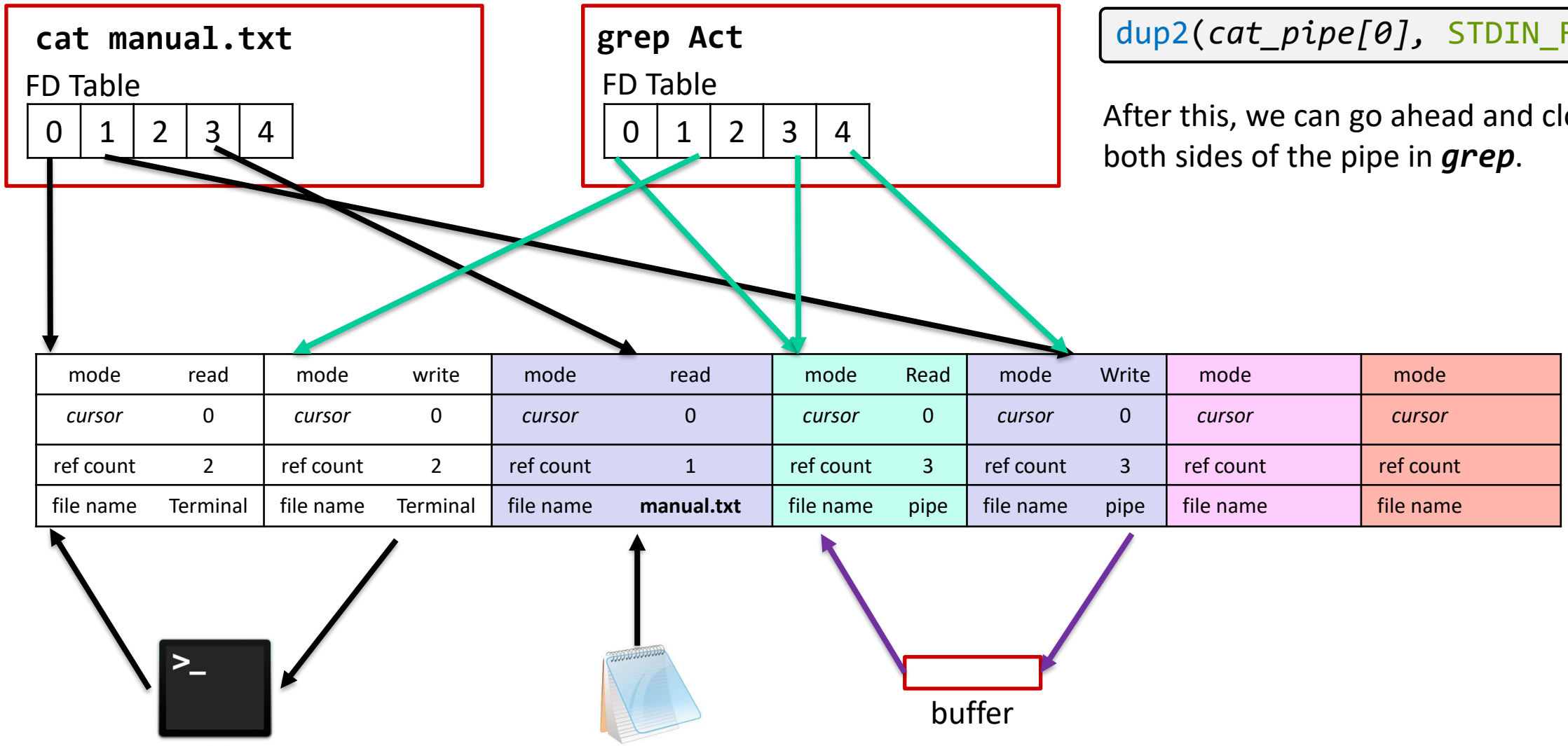


grep must read from the pipe, and as the pipe is inherited via a fork, it could have access to both read and write portions.

We'll have to redirect where **STDIN** refers to within the grep process.

```
dup2(cat_pipe[0], STDIN_FILENO);
```

cat manual.txt | grep Act | uniq




```
dup2(cat_pipe[0], STDIN_FILENO);
```

After this, we can go ahead and close both sides of the pipe in ***grep***.

cat manual.txt | grep Act | uniq

cat manual.txt

FD Table

0	1	2	3	4

grep Act

FD Table

0	1	2	3	4

```
dup2(cat_pipe[0], STDIN_FILENO);
```

After this, we can go ahead and close both sides of the pipe in ***grep***.

Check out our first loop of pipes in red!

mode	read	mode	write	mode	read	mode	Read	mode	Write	mode	mode
cursor	0	*cursor*	0	*cursor*	0	*cursor*	0	*cursor*	0	*cursor*	*cursor*
ref count	2	ref count	2	ref count	1	ref count	2	ref count	2	ref count	ref count
file name	Terminal	file name	Terminal	file name	**manual.txt**	file name	pipe	file name	pipe	file name	file name

buffer

`cat manual.txt | grep Act | uniq`

`cat manual.txt`

FD Table

0	1	2	3	4

`grep Act`

FD Table

0	1	2	3	4

WAIT! **`grep`** must also redirect ***STDOUT*** to the write end of a pipe it must share with **`uniq`**

How else will uniq receive input from grep?

mode	read	mode	write	mode	read	mode	Read	mode	Write	mode	Read	mode	Write
cursor	0	*cursor*	0	*cursor*	0	*cursor*	0	*cursor*	0	*cursor*	0	*cursor*	0
ref count	2	ref count	2	ref count	1	ref count	2	ref count	2	ref count	1	ref count	1
file name	Terminal	file name	Terminal	file name	**manual.txt**	file name	pipe	file name	pipe	file name	pipe	file name	pipe

buffer

`cat manual.txt | grep Act | uniq`

`cat manual.txt`

FD Table

0	1	2	3	4

`grep Act`

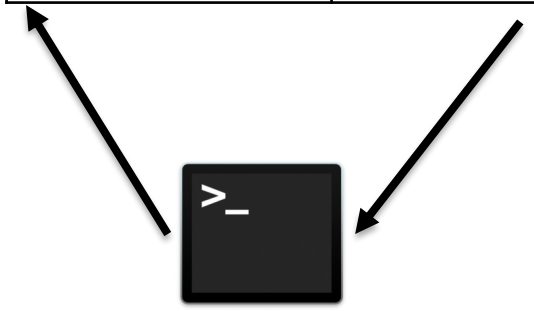
FD Table

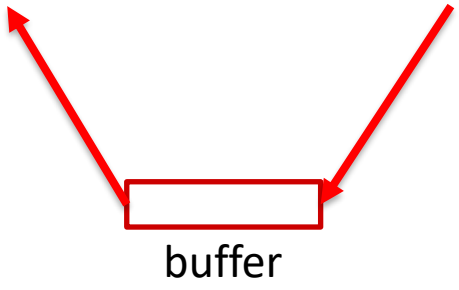
0	1	2	3	4

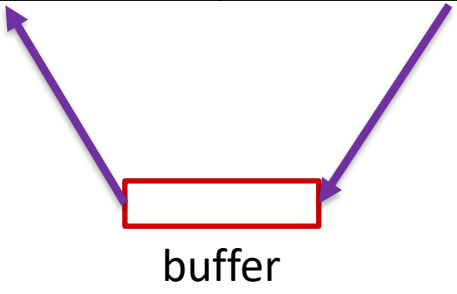
WAIT! `grep` must also redirect ***STDOUT*** to the write end of a pipe it must share with **`uniq`**

```
dup2(grep_pipe[1], STDOUT_FILENO);
```

mode	read	mode	write	mode	read	mode	Read	mode	Write	mode	Read	mode	Write
cursor	0	*cursor*	0	*cursor*	0	*cursor*	0	*cursor*	0	*cursor*	0	*cursor*	0
ref count	2	ref count	1	ref count	1	ref count	1	ref count	1	ref count	1	ref count	1
file name	Terminal	file name	Terminal	file name	**manual.txt**	file name	pipe	file name	pipe	file name	pipe	file name	pipe







Poll Everywhere

pollev.com/tqm

```
pipe(&grep_fds); // A ←
pid_t cat_pid = fork();
pipe(&grep_fds); // B ←
if(cat_pid == 0){
   // do cat stuff
   // maybe do some pipe stuff?
}
pipe(&grep_fds); // C ←
pid_t grep_pid = fork();
pipe(&grep_fds); // D ←
if(grep_pid == 0){
   // do grep stuff
   // maybe do some pipe stuff?
}
pipe(&grep_fds); // E ←
pid_t uniq_pid = fork();
pipe(&grep_fds); // F ←
if(uniq_pid == 0){
   // do uniq stuff
 }
```

`cat manual.txt | grep Act | uniq`

Where is the ***best place*** to put a pipe, so both grep and uniq can write and read, respectively?

*Yes, this is a completely different pipe from the one shared by cat and grep

Poll Everywhere

pollev.com/tqm

```
pipe(&grep_fds); // A ←
pid_t cat_pid = fork();
pipe(&grep_fds); // B ←
if(cat_pid == 0){
   // do cat stuff
   // maybe do some pipe stuff?
}
pipe(&grep_fds); // C ←
pid_t grep_pid = fork();
pipe(&grep_fds); // D ←
if(grep_pid == 0){
   // do grep stuff
   // maybe do some pipe stuff?
}
pipe(&grep_fds); // E ←
pid_t uniq_pid = fork();
pipe(&grep_fds); // F ←
if(uniq_pid == 0){
   // do uniq stuff
 }
```

`cat manual.txt | grep Act | uniq`

F: This creates two sperate pipes, in the `uniq` & parent process only. This pipe does not exist in the FD Table of `grep`! No way to communicate.

Poll Everywhere

pollev.com/tqm

```
pipe(&grep_fds); // A ←
pid_t cat_pid = fork();
pipe(&grep_fds); // B ←
if(cat_pid == 0){
   // do cat stuff
   // maybe do some pipe stuff?
}
pipe(&grep_fds); // C ←
pid_t grep_pid = fork();
pipe(&grep_fds); // D ←
if(grep_pid == 0){
   // do grep stuff
   // maybe do some pipe stuff?
}
pipe(&grep_fds); // E ←
pid_t uniq_pid = fork();
pipe(&grep_fds); // F ←
if(uniq_pid == 0){
   // do uniq stuff
 }
```

`cat manual.txt | grep Act | uniq`

E: This creates one pipe, that is shared by both the parent process and uniq! However, still inaccessible by both uniq and grep.

Poll Everywhere

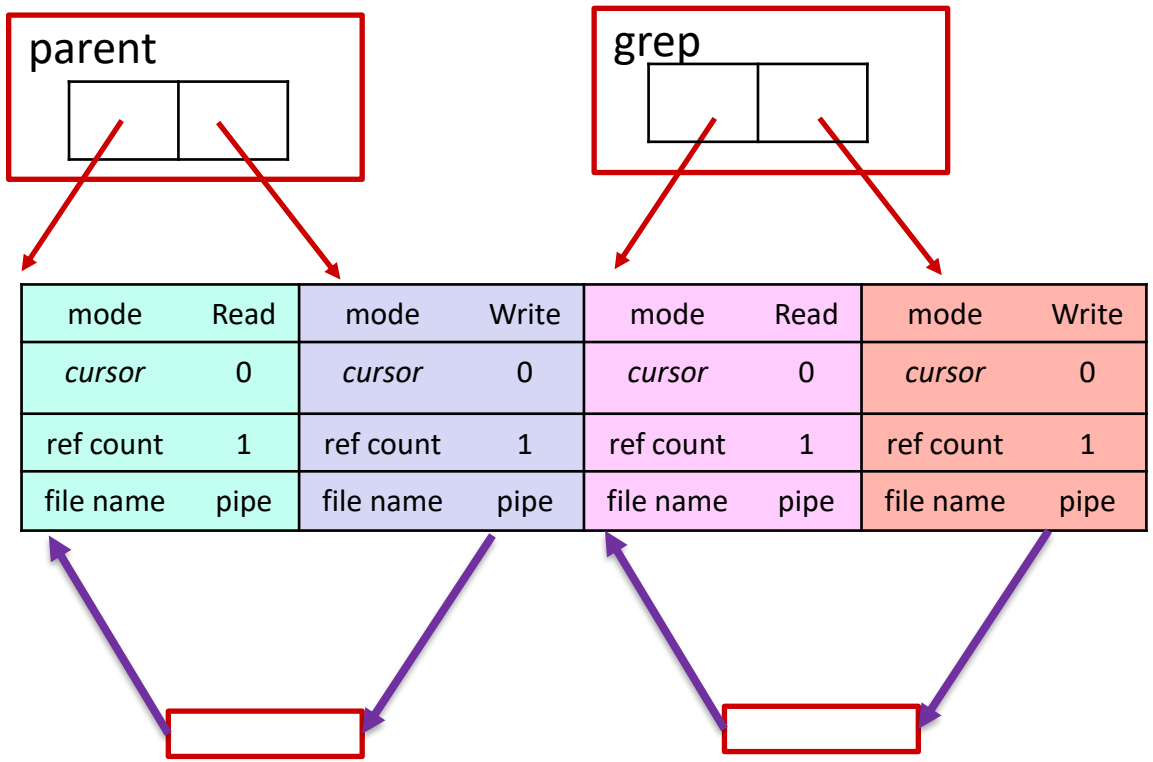
pollev.com/tqm

```
pipe(&grep_fds); // A ←
pid_t cat_pid = fork();
pipe(&grep_fds); // B ←
if(cat_pid == 0){
   // do cat stuff
   // maybe do some pipe stuff?
}
pipe(&grep_fds); // C ←
pid_t grep_pid = fork();
pipe(&grep_fds); // D ←
if(grep_pid == 0){
   // do grep stuff
   // maybe do some pipe stuff?
}
pipe(&grep_fds); // E ←
pid_t uniq_pid = fork();
pipe(&grep_fds); // F ←
if(uniq_pid == 0){
   // do uniq stuff
 }
```

`cat manual.txt | grep Act | uniq`

D: This creates two separate pipes, one in the parent and one in the grep process. However, still inaccessible by both uniq and grep. Why…

***Which of these will uniq inherit*?**



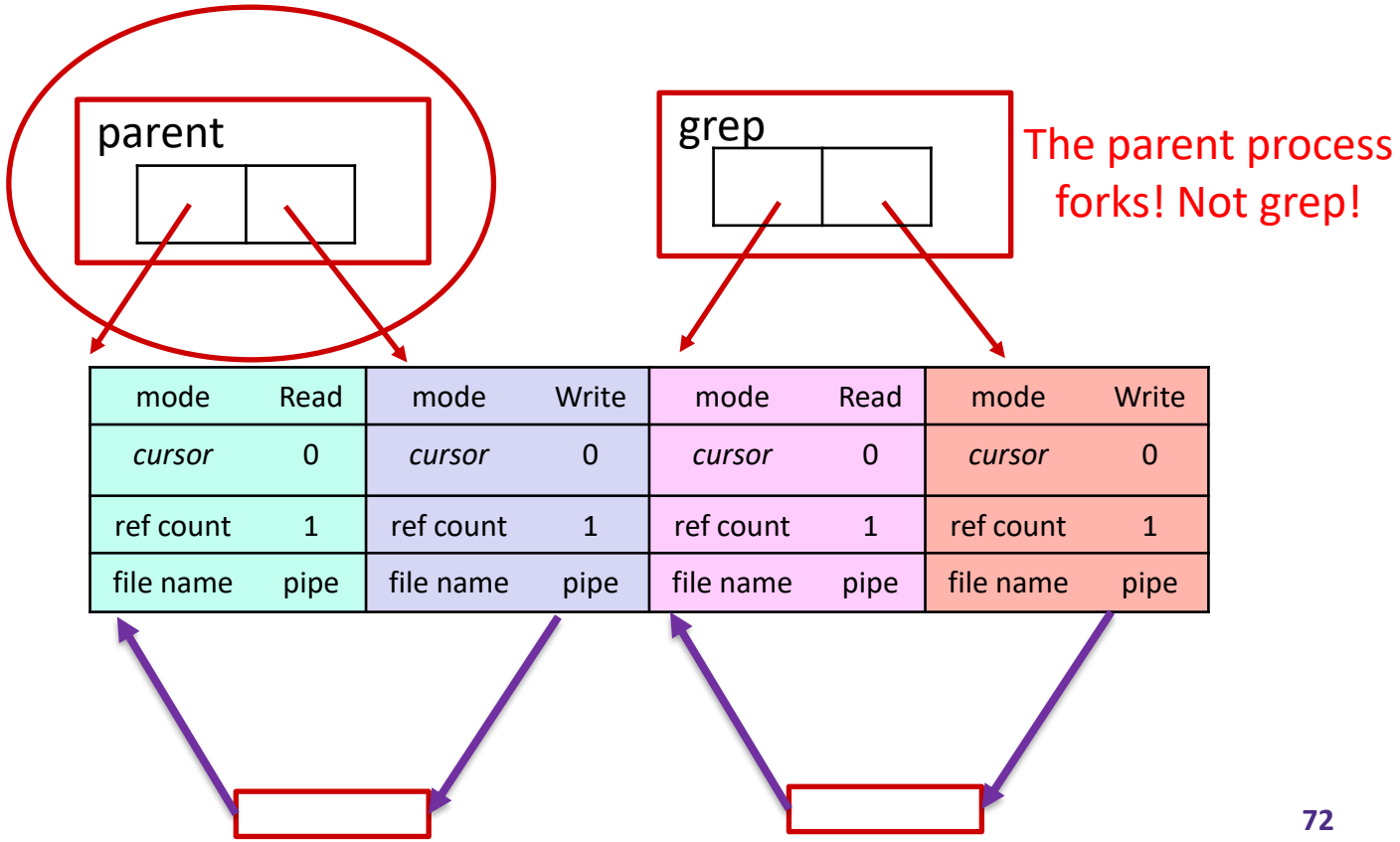
Poll Everywhere

pollev.com/tqm

```
pipe(&grep_fds); // A ←
pid_t cat_pid = fork();
pipe(&grep_fds); // B ←
if(cat_pid == 0){
   // do cat stuff
   // maybe do some pipe stuff?
}
pipe(&grep_fds); // C ←
pid_t grep_pid = fork();
pipe(&grep_fds); // D ←
if(grep_pid == 0){
   // do grep stuff
   // maybe do some pipe stuff?
}
pipe(&grep_fds); // E ←
pid_t uniq_pid = fork();
pipe(&grep_fds); // F ←
if(uniq_pid == 0){
   // do uniq stuff
 }
```

`cat manual.txt | grep Act | uniq`

D: This creates two separate pipes, one in the parent and one in the grep process. However, still inaccessible by both uniq and grep. Why…***Which of these will uniq inherit***?



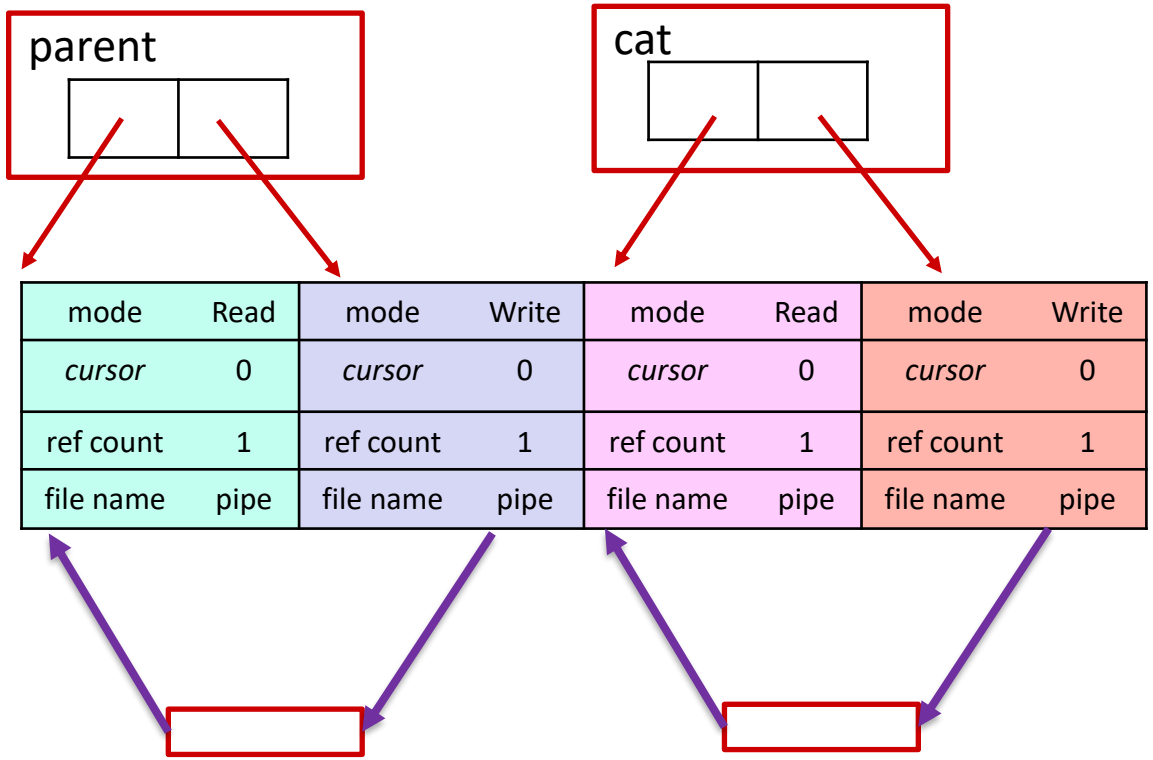
Poll Everywhere

pollev.com/tqm

```
pipe(&grep_fds); // A ←
pid_t cat_pid = fork();
pipe(&grep_fds); // B ←
if(cat_pid == 0){
   // do cat stuff
   // maybe do some pipe stuff?
}
pipe(&grep_fds); // C ←
pid_t grep_pid = fork();
pipe(&grep_fds); // D ←
if(grep_pid == 0){
   // do grep stuff
   // maybe do some pipe stuff?
}
pipe(&grep_fds); // E ←
pid_t uniq_pid = fork();
pipe(&grep_fds); // F ←
if(uniq_pid == 0){
   // do uniq stuff
 }
```

`cat manual.txt | grep Act | uniq`

B: Will create a pipe in the parent accessible by grep and uniq BUT also makes a pipe that cat must deal with.



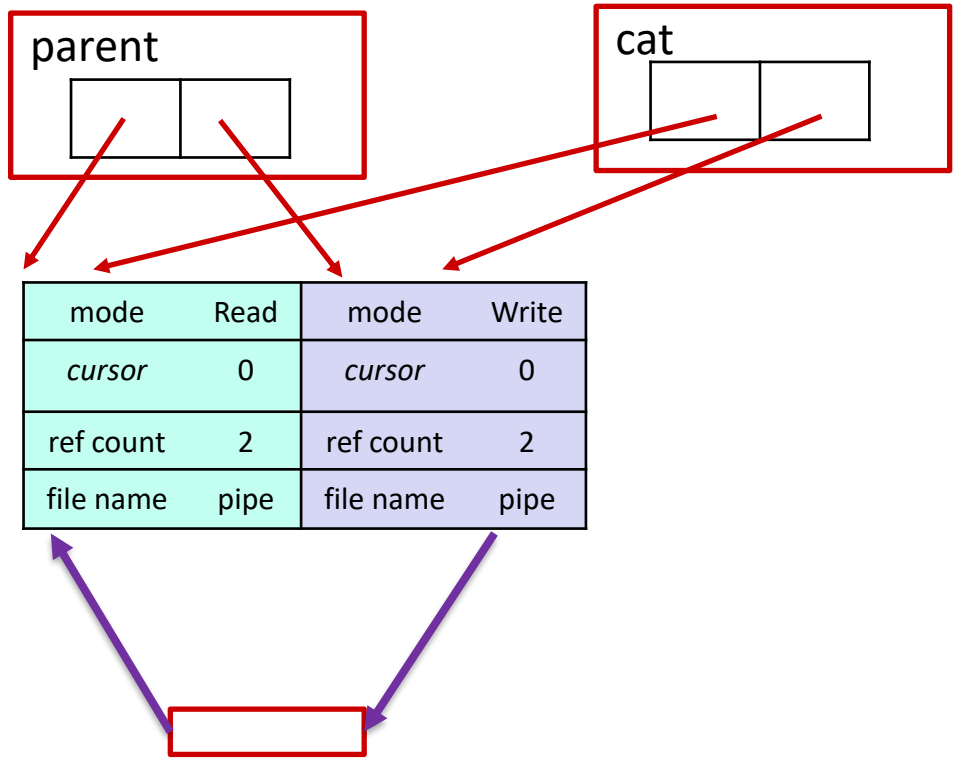
Poll Everywhere

pollev.com/tqm

```
pipe(&grep_fds); // A ←
pid_t cat_pid = fork();
pipe(&grep_fds); // B ←
if(cat_pid == 0){
   // do cat stuff
   // maybe do some pipe stuff?
}
pipe(&grep_fds); // C ←
pid_t grep_pid = fork();
pipe(&grep_fds); // D ←
if(grep_pid == 0){
   // do grep stuff
   // maybe do some pipe stuff?
}
pipe(&grep_fds); // E ←
pid_t uniq_pid = fork();
pipe(&grep_fds); // F ←
if(uniq_pid == 0){
   // do uniq stuff
 }
```

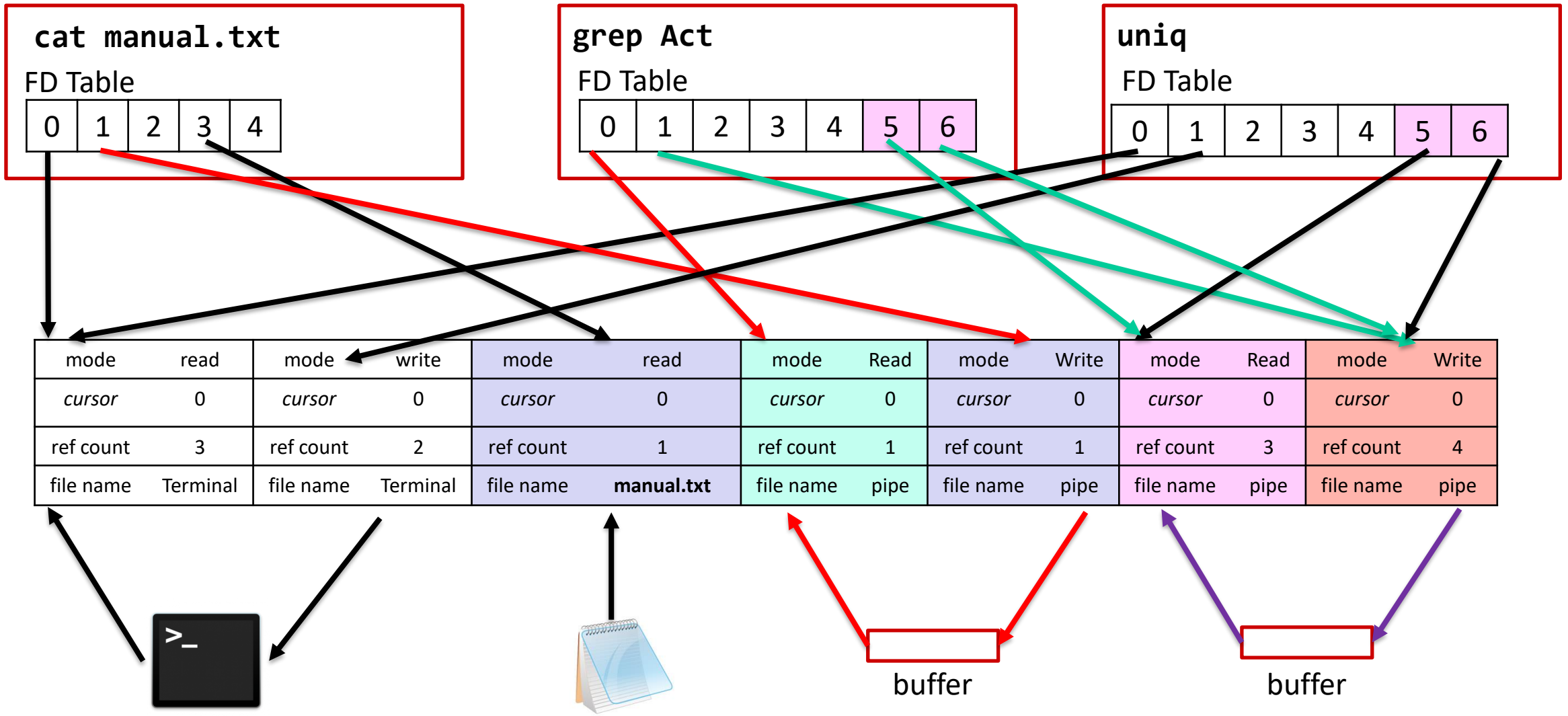
`cat manual.txt | grep Act | uniq`

A: Will create a pipe in the parent accessible by grep and uniq BUT also makes a pipe that cat must deal with.



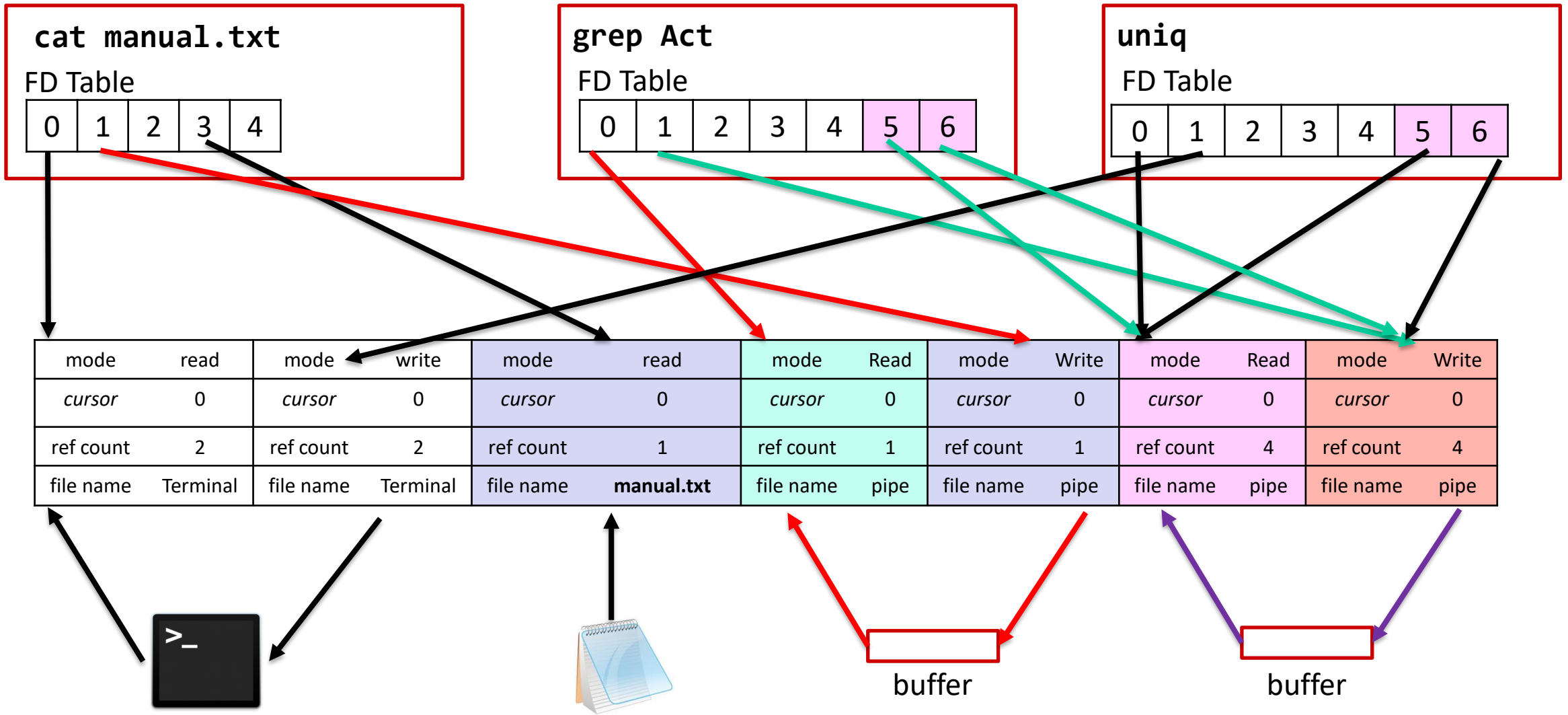
`cat manual.txt | grep Act | uniq`

Finally, uniq reads from pipe shared with 'grep' via dup2



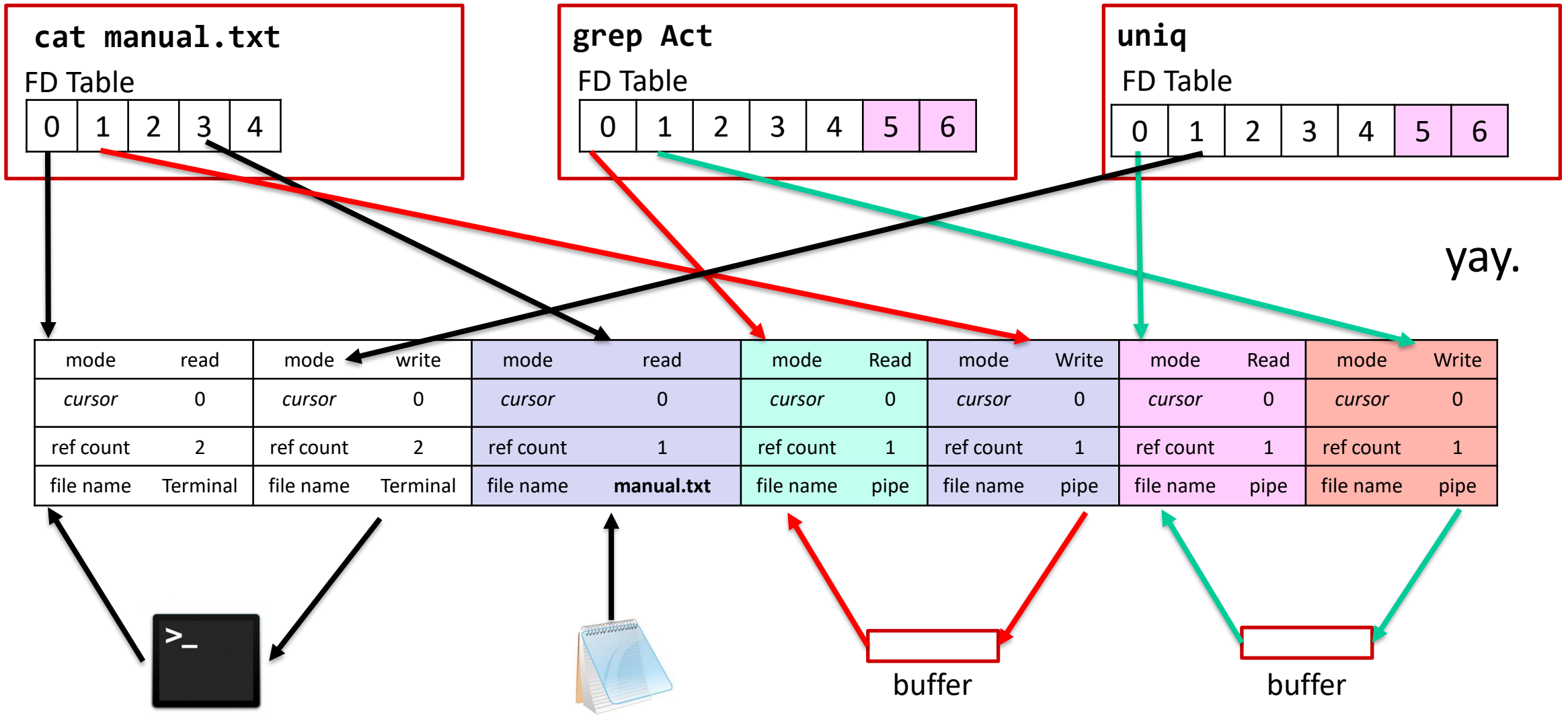
`cat manual.txt | grep Act | uniq`

Let's close all unnecessary FDs so we can see the beauty…



`cat manual.txt | grep Act | uniq`

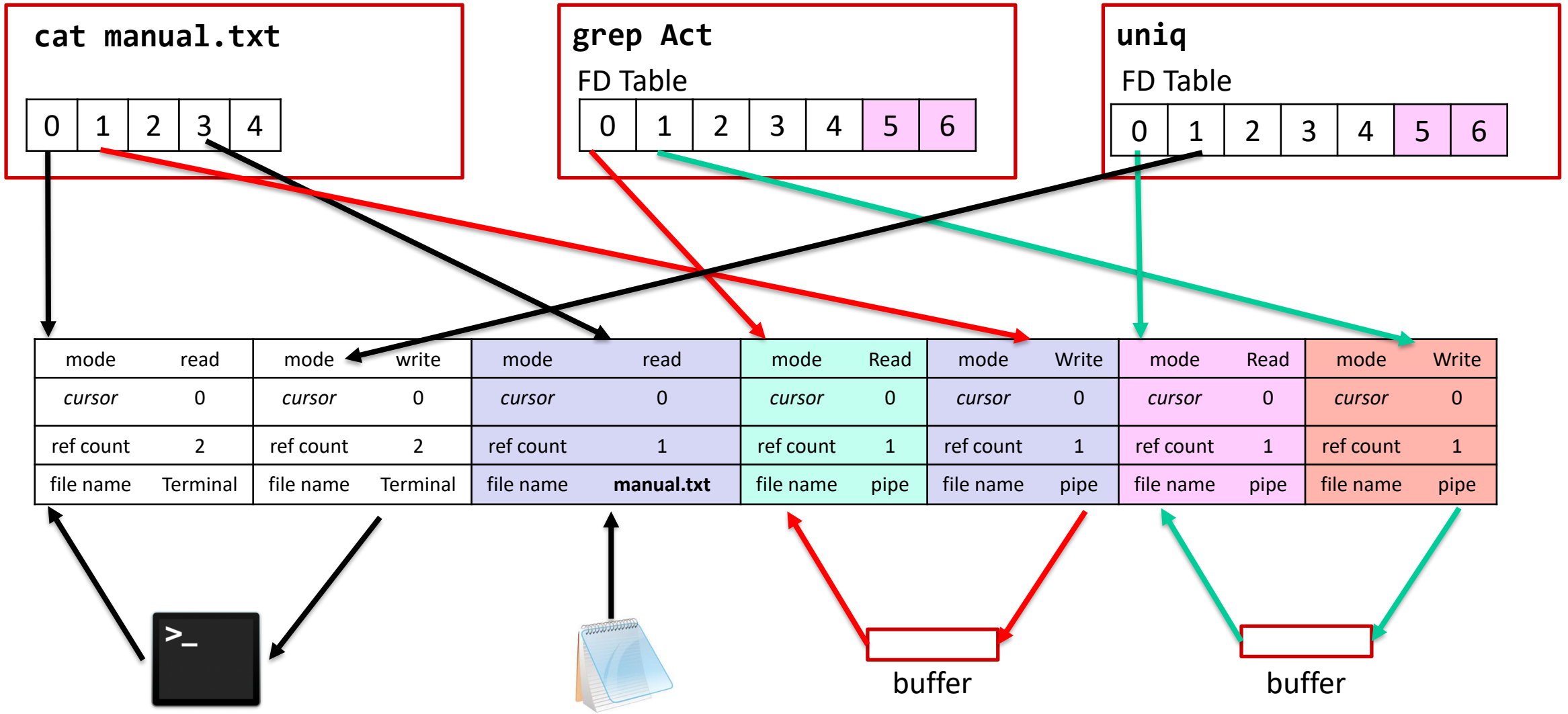
Let's close all unnecessary FDs so we can see the beauty…



Poll Everywhere

Discuss

Why doesn't uniq need to redirect it's STDOUT?



Poll Everywhere

Discuss

Why doesn't uniq need to redirect it's STDOUT?

cat manual.txt

0	1	2	3	4

grep Act

FD Table

0	1	2	3	4	5	6

uniq

FD Table

0	1	2	3	4	5	6

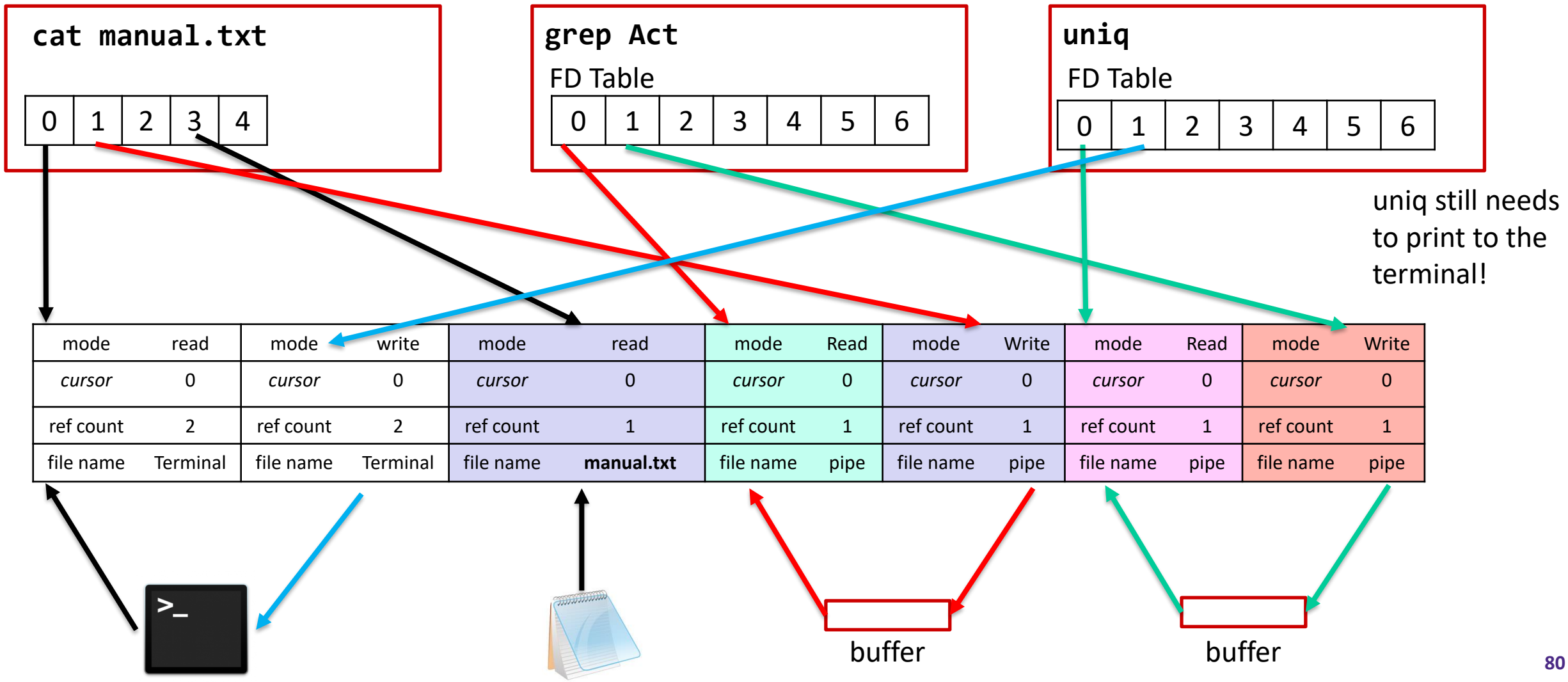
uniq still needs to print to the terminal!

mode	read	mode	write	mode	read	mode	Read	mode	Write	mode	Read	mode	Write
cursor	0	*cursor*	0	*cursor*	0	*cursor*	0	*cursor*	0	*cursor*	0	*cursor*	0
ref count	2	ref count	2	ref count	1	ref count	1	ref count	1	ref count	1	ref count	1
file name	Terminal	file name	Terminal	file name	**manual.txt**	file name	pipe	file name	pipe	file name	pipe	file name	pipe

buffer

buffer

Let's see it in code! Cool.



Poll Everywhere

pollev.com/tqm

```
close(cat_pipe[READ_END]);
//close(grep_pipe[WRITE_END]);

pid_t uniq_pid = fork();
if(uniq_pid == 0) {
    dup2(grep_pipe[READ_END],STDIN_FILENO);
    close(grep_pipe[READ_END]);
    execvp(uniq_argv[0], uniq_argv);
    _exit(EXIT_FAILURE);
}


wait(NULL);
wait(NULL);
wait(NULL);
```

What happens if you forget to close a write portion of the pipe, before waiting in the parent?

Forgetting to Close Pipes

```
close(cat_pipe[READ_END]);
//close(grep_pipe[WRITE_END]);

pid_t uniq_pid = fork();
if(uniq_pid == 0) {
    dup2(grep_pipe[READ_END],STDIN_FILENO);
    close(grep_pipe[READ_END]);
    execvp(uniq_argv[0], uniq_argv);
    _exit(EXIT_FAILURE);
}


wait(NULL);
wait(NULL);
wait(NULL);
```

uniq must read from STDIN ***but it does not stop reading from STDIN until it receives an EOF!***

The bigger issue is in the parent as that tends to be the one which has access to all write ends of the pipe. Make sure to close them as soon as you don't need them.

FDs are closed when a program is terminated. The trick is to make sure it terminates and doesn't hang!

Poll Everywhere

Discuss

```
close(cat_pipe[READ_END]);

pid_t uniq_pid = fork();

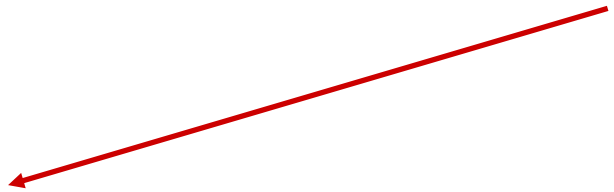
if(uniq_pid == 0) {
    dup2(grep_pipe[READ_END],STDIN_FILENO);
    close(grep_pipe[READ_END]);
    execvp(uniq_argv[0], uniq_argv);
    _exit(EXIT_FAILURE);
}

close(grep_pipe[WRITE_END]);

wait(NULL);
wait(NULL);
wait(NULL);
```

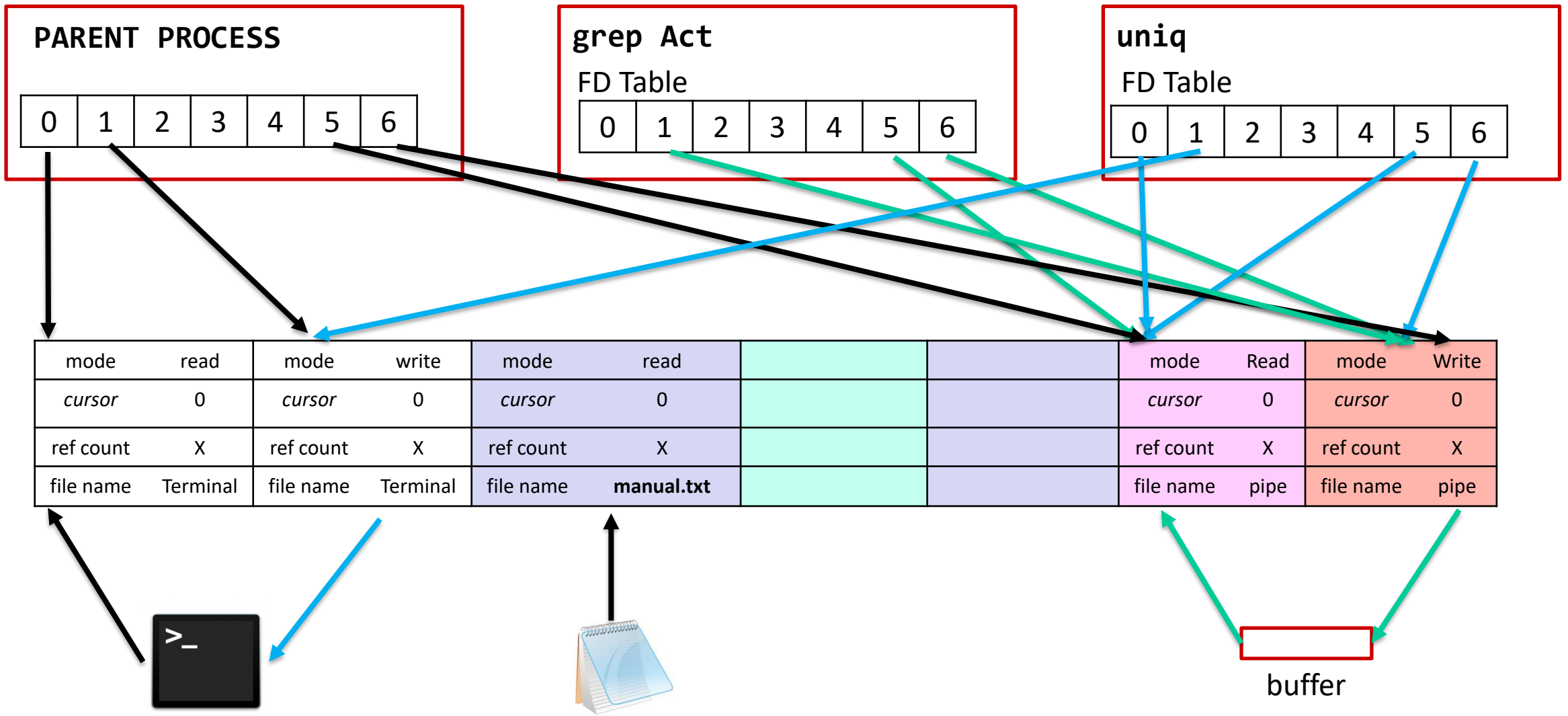
Exam Style Question:

In office hours a student attempts to fix the previous code by adding the close here. Does this work? Why or why not?



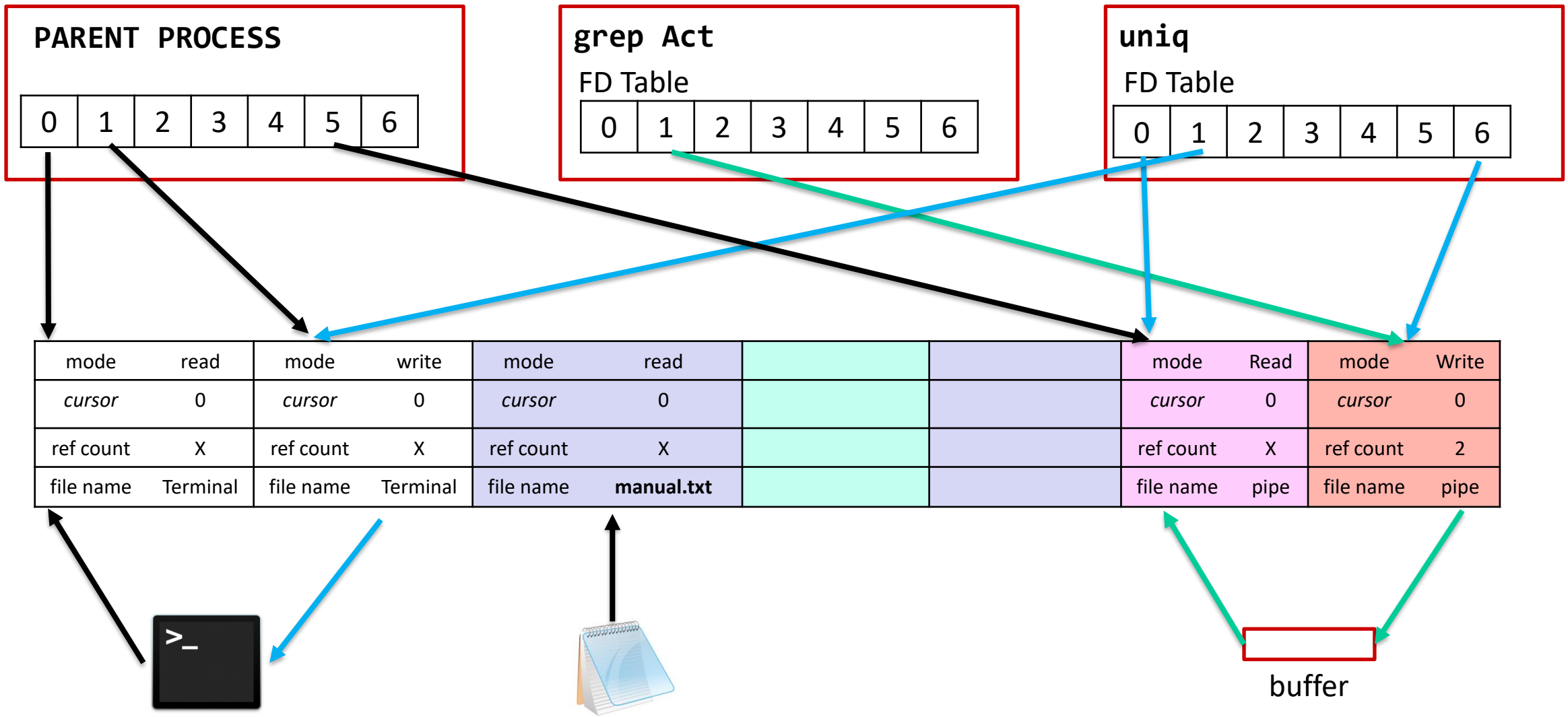
Exam-Style Question Walkthrough

We start off with grep and uniq having both references to the pipe!

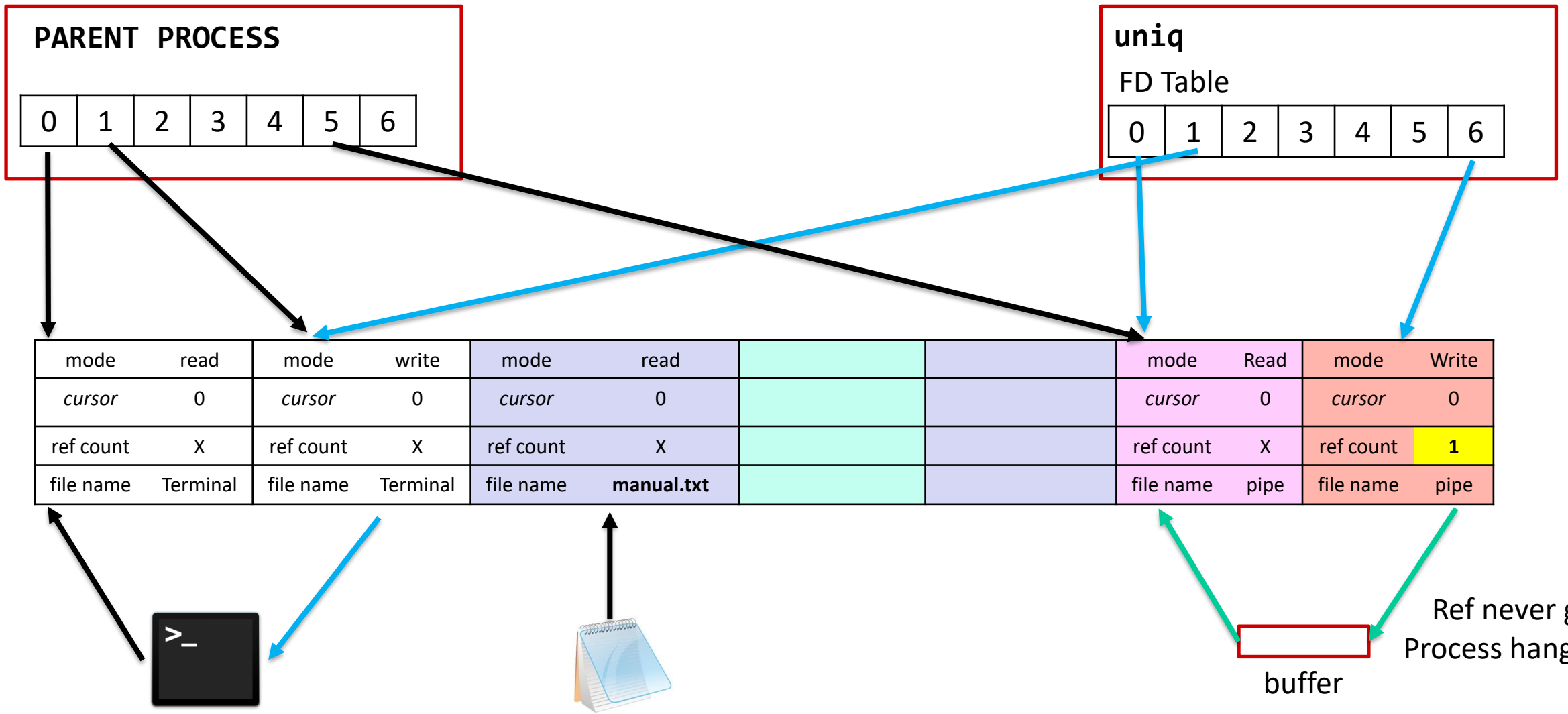


Exam-Style Question Walkthrough

Then Grep will terminate…



Exam-Style Question Walkthrough



Lecture Outline

- Unix Commands & Controls
- Review
  - File Descriptors
  - File Table
  - Open File Table
- Pipes and Dup
- **pipe2**

pipe2

```
int pipe2(int pipefd[2], int flags);
```

- Still creates a pipe, like `pipe`, but we can now specify behavior!
- Flags
  - O_CLOEXEC, your new friend Also a flag for open!
  - Enables the close-on-exec flag for the file descriptors
  - ***FDs with this flag enabled will be closed when that process execs***
    - Only closes the references from the ***exec'ing*** process!
    - File descriptors that are ***dup2'd*** with these FDs are not closed!
- Requires `#define _GNU_SOURCE`
  - Check the man page!
  - `pipe2()` is Linux-specific

pipe2

```
int pipe2(int pipefd[2], int flags);
```

- Here's an equivalent macro, for those not on Linux machines.

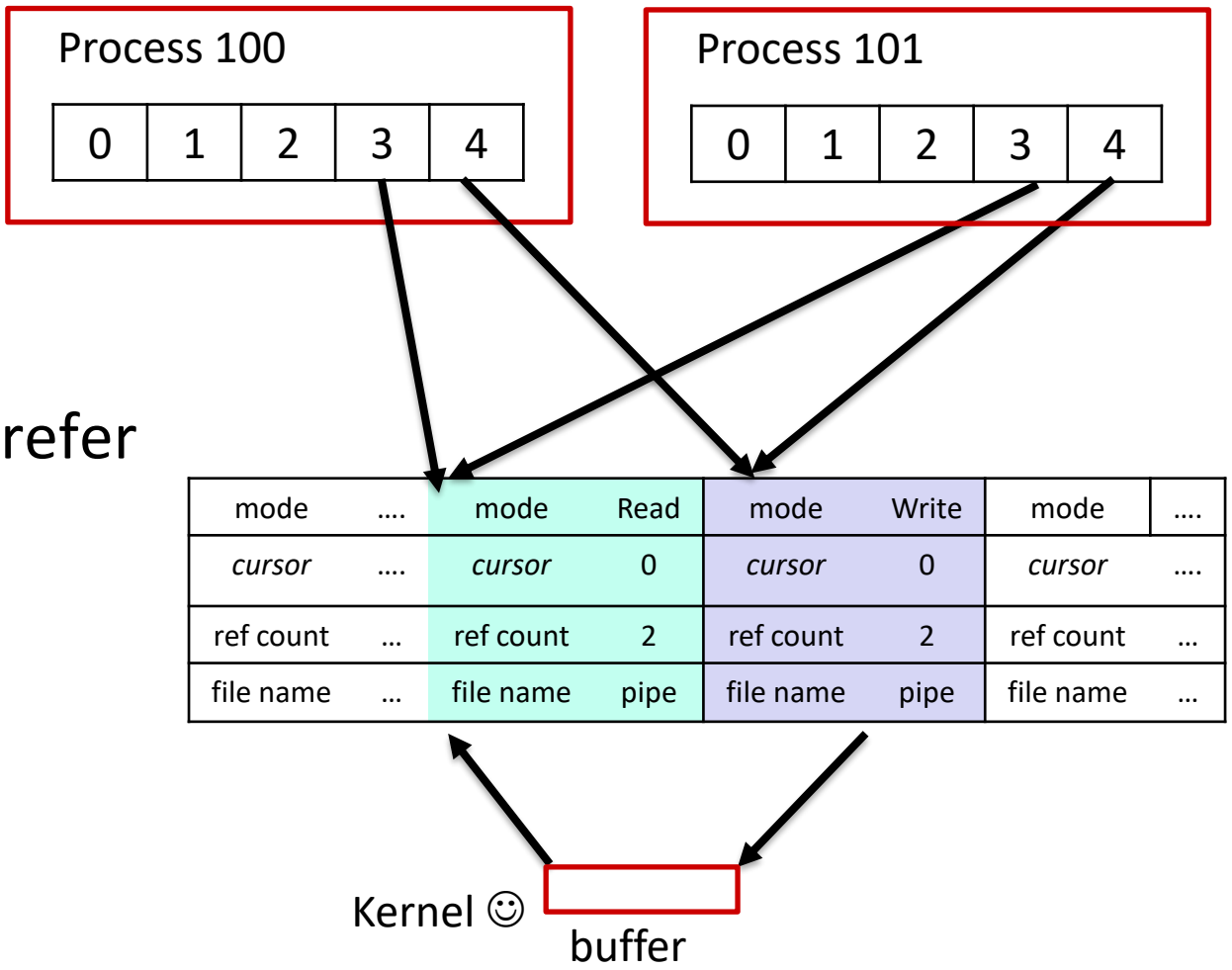
```
#define pipe2(FD, FLAG) \
pipe((FD)); \
fcntl((FD)[0], F_SETFD, FLAG); \
fcntl((FD)[1], F_SETFD, FLAG);
```

O_CLOEXEC Behavior

```
int pipe_fds[2];
pipe2(&pipe_fds, O_CLOEXEC);
pid_t cat_pid = fork();

if(cat_pid == 0){
    execvp(…);
}
// parent does some stuff.
```

- Prior to the execvp, both processes refer to the same pipe!

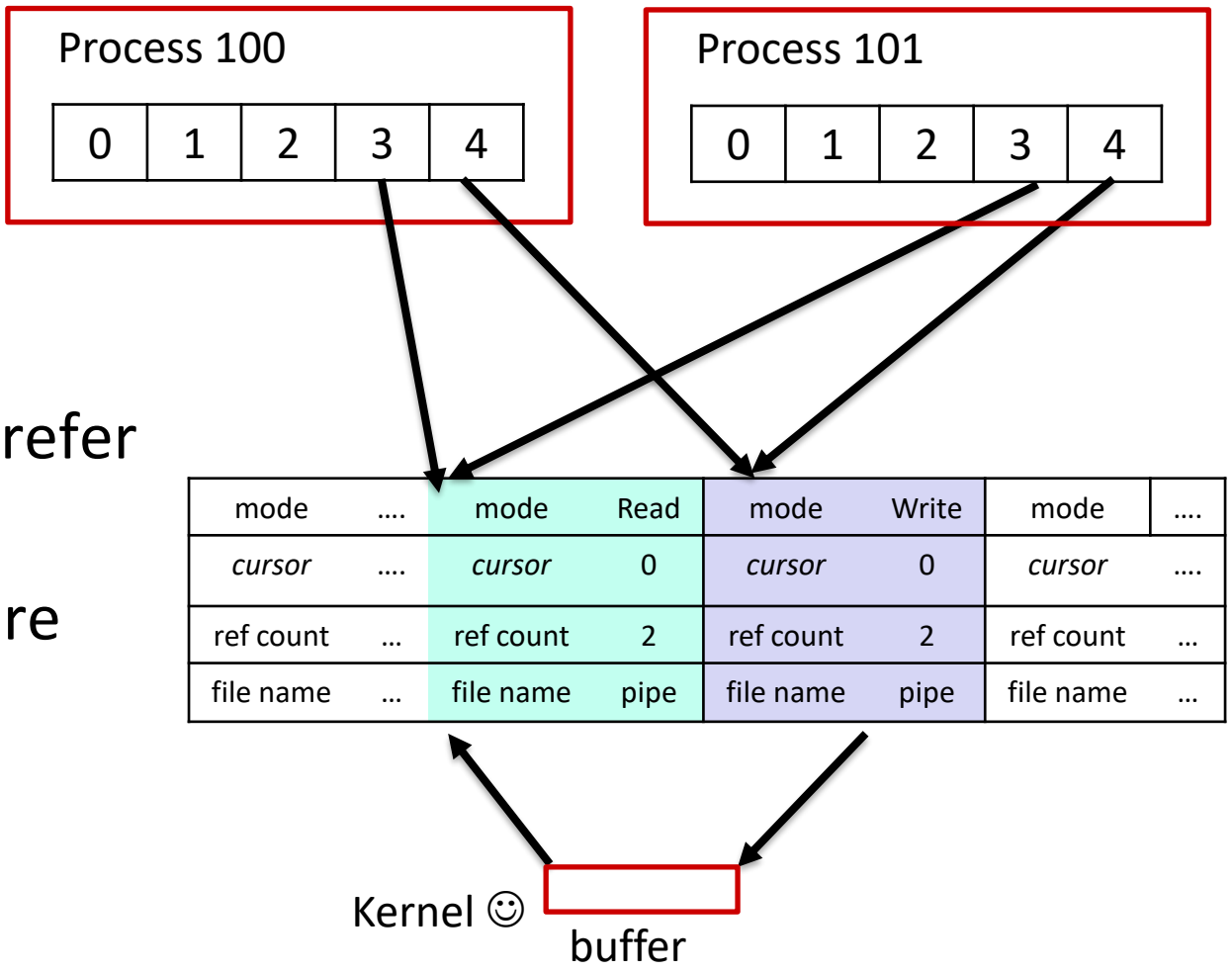


O_CLOEXEC Behavior

```
int pipe_fds[2];
pipe2(&pipe_fds, O_CLOEXEC);
pid_t cat_pid = fork();

if(cat_pid == 0){
    execvp(…);
}
// parent does some stuff.
```

- Prior to the execvp, both processes refer to the same pipe!
- Once the child execs, the pipe_fds are closed!

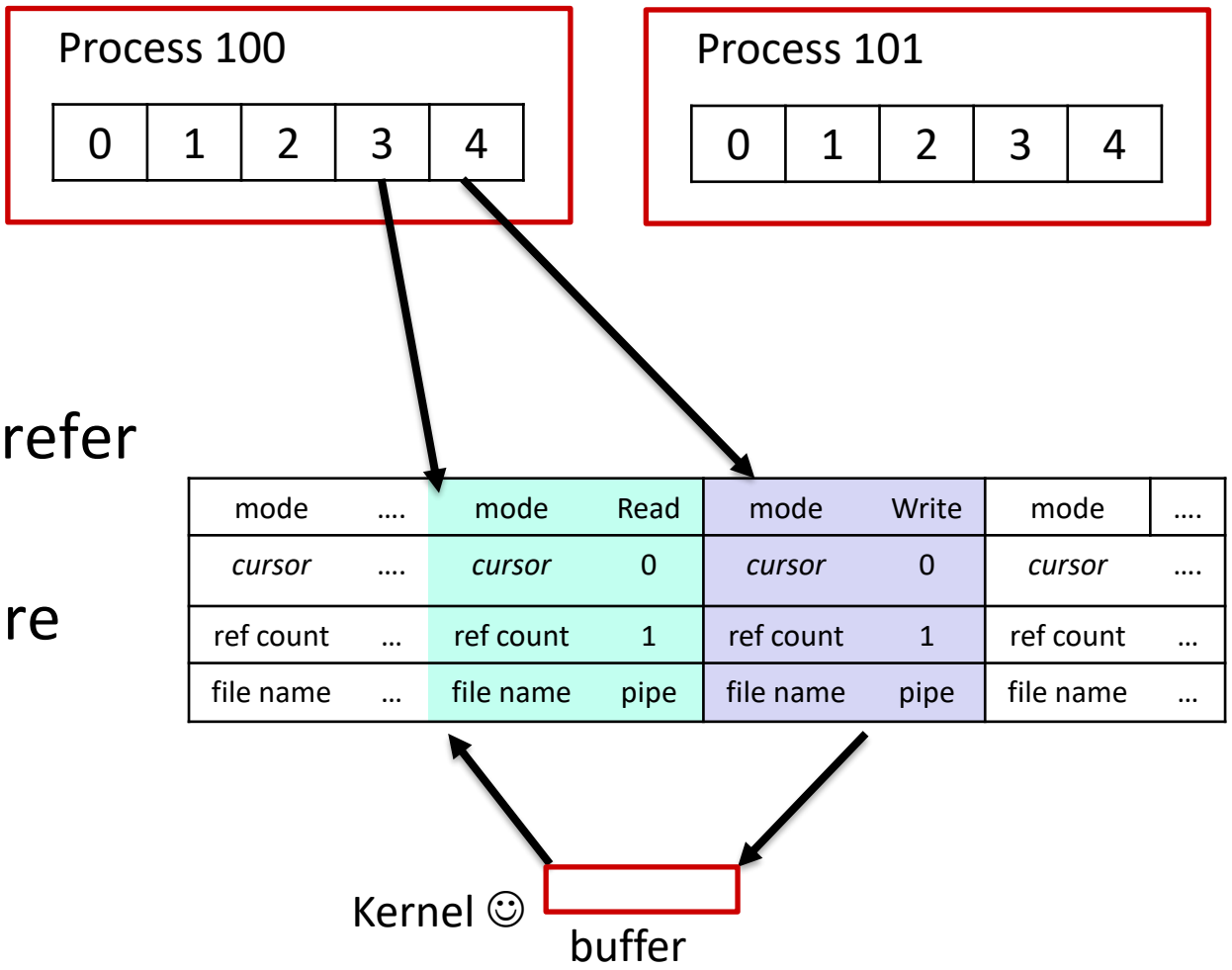


O_CLOEXEC Behavior

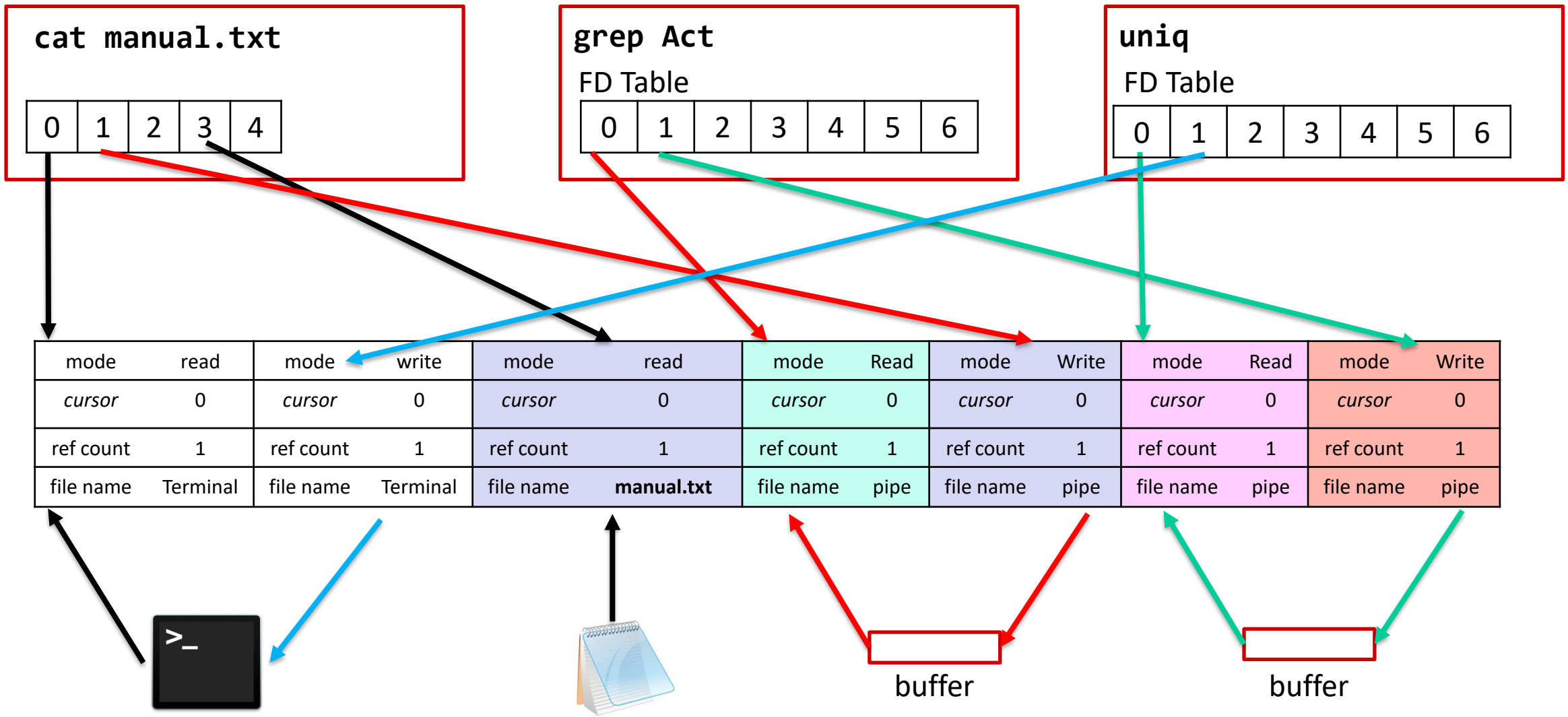
```
int pipe_fds[2];
pipe2(&pipe_fds, O_CLOEXEC);
pid_t cat_pid = fork();

if(cat_pid == 0){
    execvp(…);
}
// parent does some stuff.
```

- Prior to the execvp, both processes refer to the same pipe!
- Once the child execs, the pipe_fds are closed!



Let's see how pipe2 changes our code…



If time, how would we implement these?

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

- **`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`**

If time, how would we implement these?

- To use `<` and `>`, you would have to open these files on behalf of the executable, and then dup2 STDIN or STDOUT.

```
cat manual.txt > copy_manual.txt
```

Here, the output from *cat* that would normally go to STDOUT, now needs to be written to this new file that we must make or ***clobber.***

If it already exists, we just overwrite what is there.

Poll Everywhere

pollev.com/tqm

`cat manual.txt > copy_manual.txt`

To make this a possibility, what should the arguments to open be? Check the ***man*** Page…

```
char *output = "copy_manual.txt";

int copy_fd = open(output, ???????, 0644);
```

"Here, the output from *cat* that would normally go to STDOUT,
now needs to be written to this new file, ***we must make*** or ***clobber (rewrite from scratch).***"

Poll Everywhere

pollev.com/tqm

```
cat manual.txt > copy_manual.txt
```

To make this a possibility, what should the arguments to open be? Check the ***man*** Page…

```
char *output = "copy_manual.txt";

int copy_fd = open(output, ???????, 0644);
```

"Here, the output from *cat* that would normally go to STDOUT,
now needs to be written to this new file, ***we must make*** or ***clobber (rewrite from scratch).***"

O_CREAT | O_TRUNC | O_WRONLY

Create the file (or open it if it exists)

Truncate the file, set its length to 0, before writing

We are only writing to it, so Write only.