More Pipes and Dup2

Computer Operating Systems, Summer 2025

Instructors: Joel Ramirez Travis McGaha

TAs: Ash Fujiyama Maya Huizar

Administrivia

- **Penn Shell**
  - Went out last Friday: Register your group on Canvas and Gradescope!
  - If you are without a partner by today at 5PM, we will automatically pair people together.
    - SO FIND SOMEONE!
- **Proj2 Milestone is due @ 11:59 pm on Monday, June 16th**
- **Project 1 Peer Evaluation is due @ 11:59 pm on Friday, June 13th**
  - **This is where your partner will critique your code…**

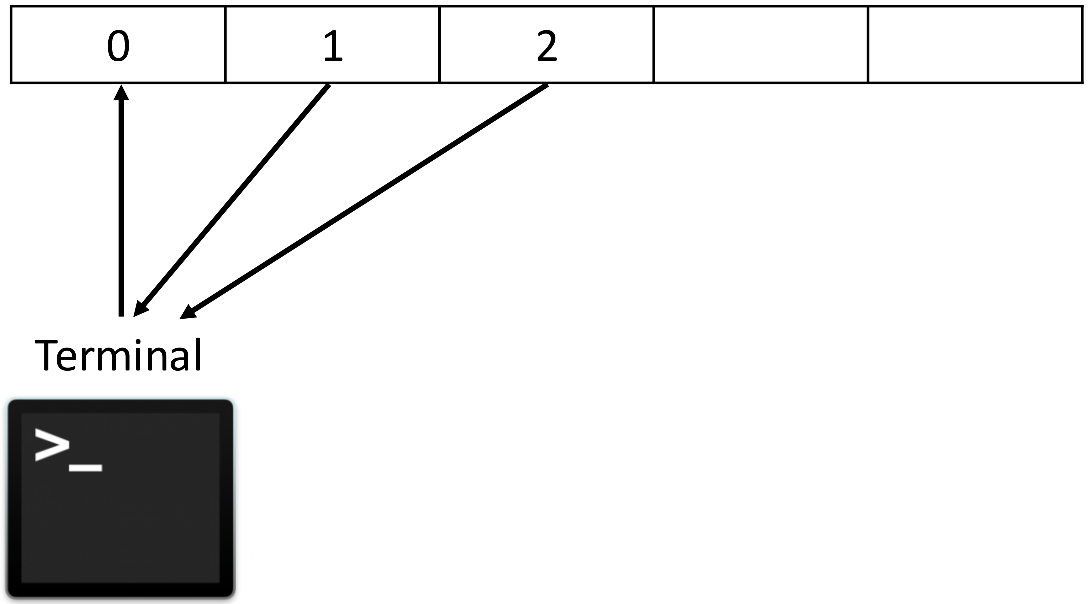
Lecture Outline

- **Quick Review**
  - **File Descriptors**
  - **File Table**
  - **Open File Table**
- Pipes and Dup
- pipe2

File Descriptor Table

- Each process has **<u>its own file descriptor table</u>** 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.

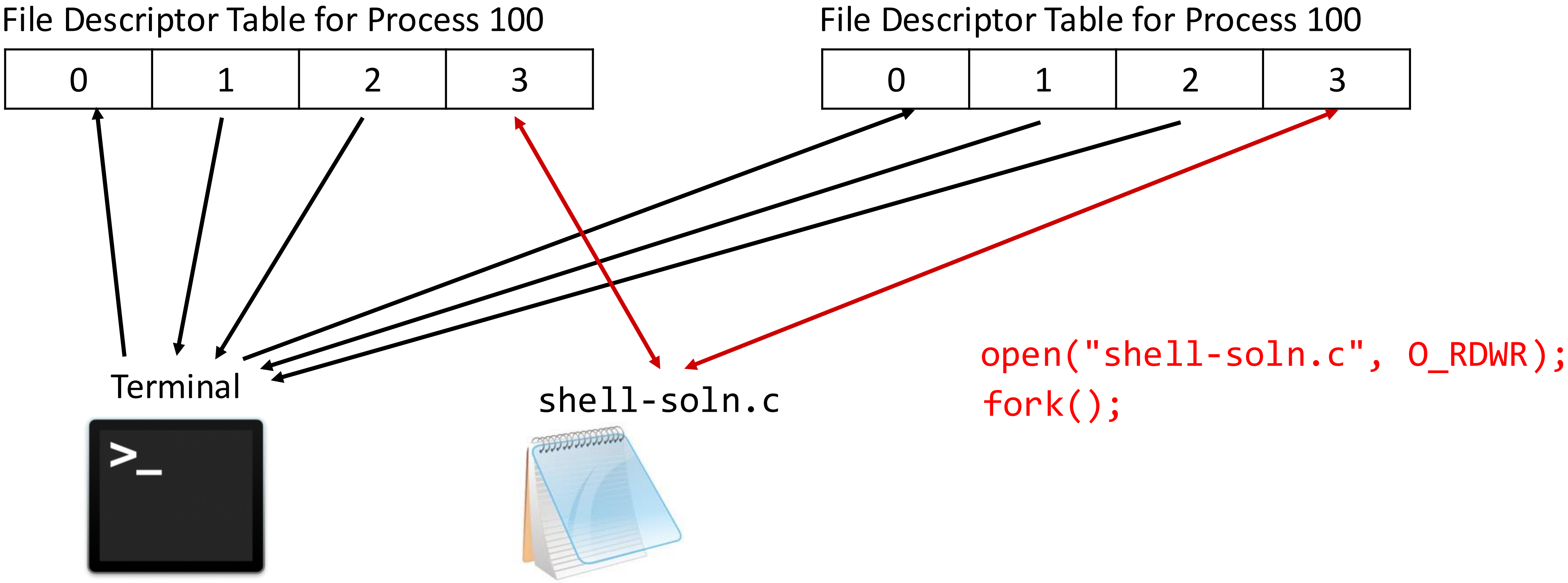
File Descriptor Table for Process 100



Not an accurate depiction.

File Descriptor Table w/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 same file reference!



File Descriptor Table for Process 100

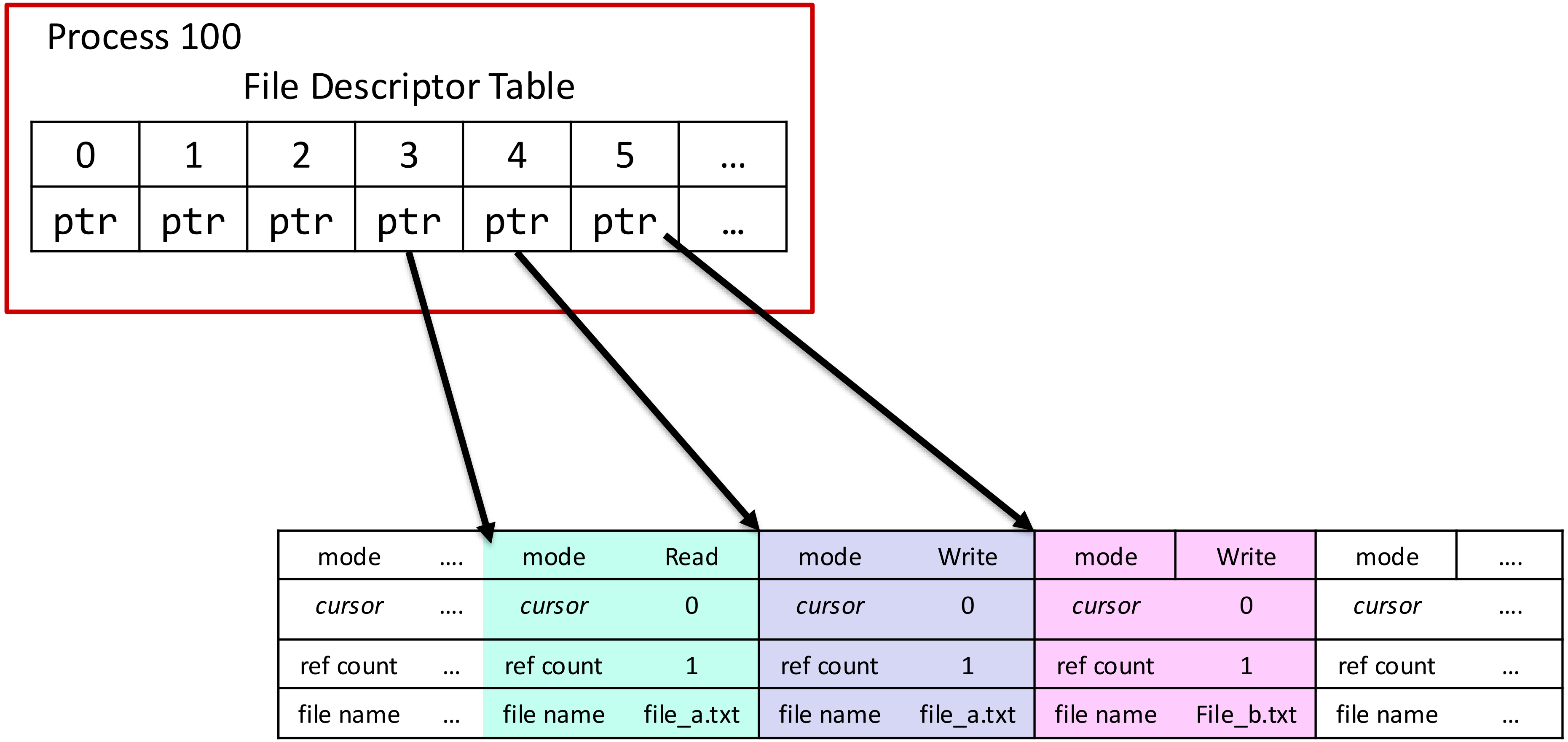
0	1	2	3

File Descriptor Table for Process 100

0	1	2	3

```
open("shell-soln.c", O_RDWR);
fork();
```

The Open File Table



Process 100

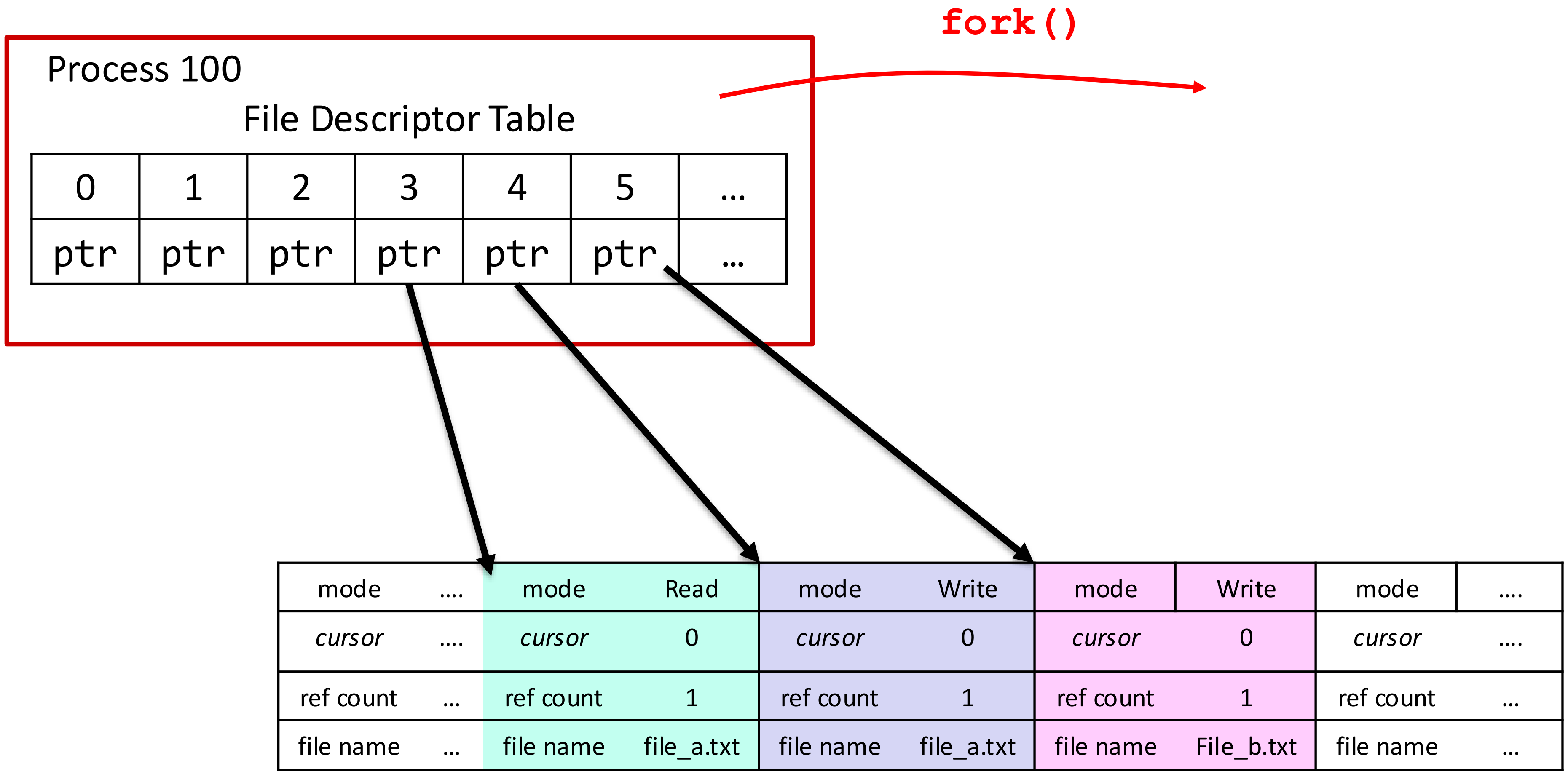
File Descriptor Table

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

mode	….	mode	Read	mode	Write	mode	Write	mode	….
cursor	….	*cursor*	0	*cursor*	0	*cursor*	0	*cursor*	….
ref count	…	ref count	1	ref count	1	ref count	1	ref count	…
file name	…	file name	file_a.txt	file name	file_a.txt	file name	File_b.txt	file name	…

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

The Open File Table



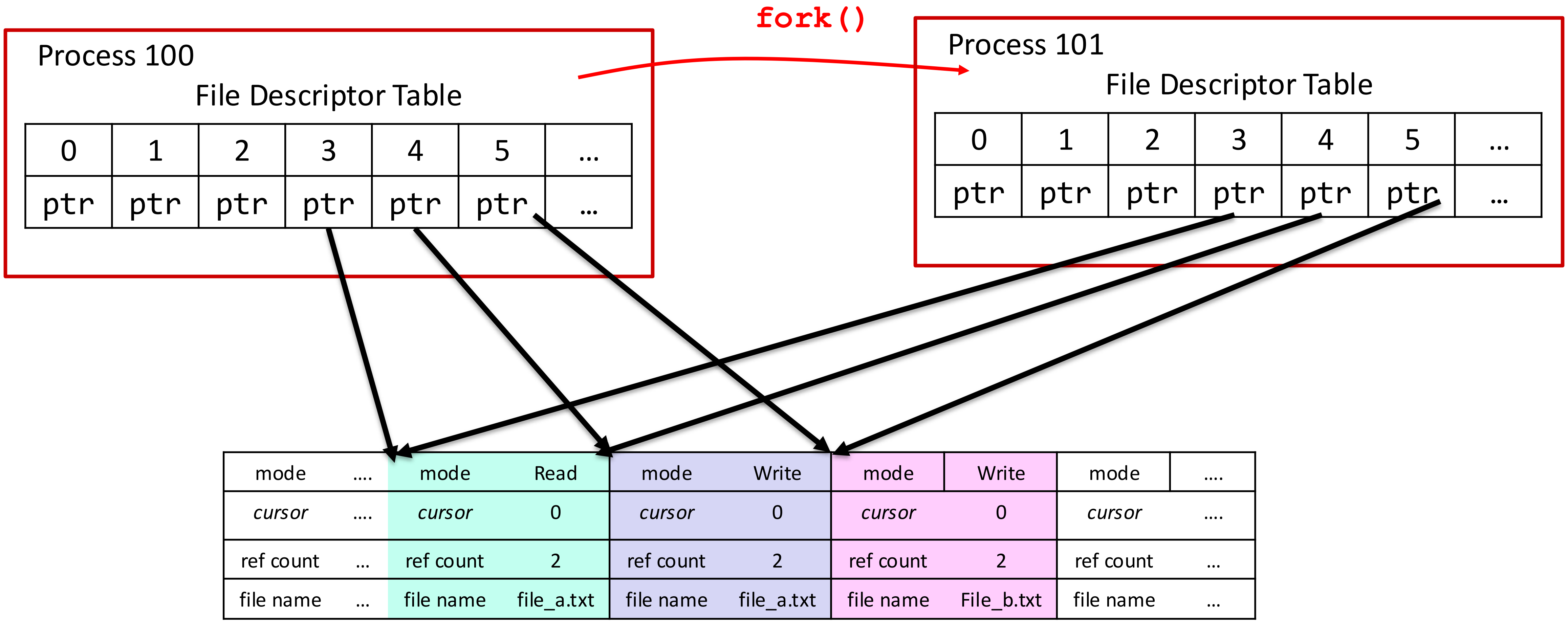
File Descriptor Table

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

mode	….	mode	Read	mode	Write	mode	Write	mode	….
cursor	….	*cursor*	0	*cursor*	0	*cursor*	0	*cursor*	….
ref count	…	ref count	1	ref count	1	ref count	1	ref count	…
file name	…	file name	file_a.txt	file name	file_a.txt	file name	File_b.txt	file name	…

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

The Open File Table



fork()

Process 100

File Descriptor Table

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

Process 101

File Descriptor Table

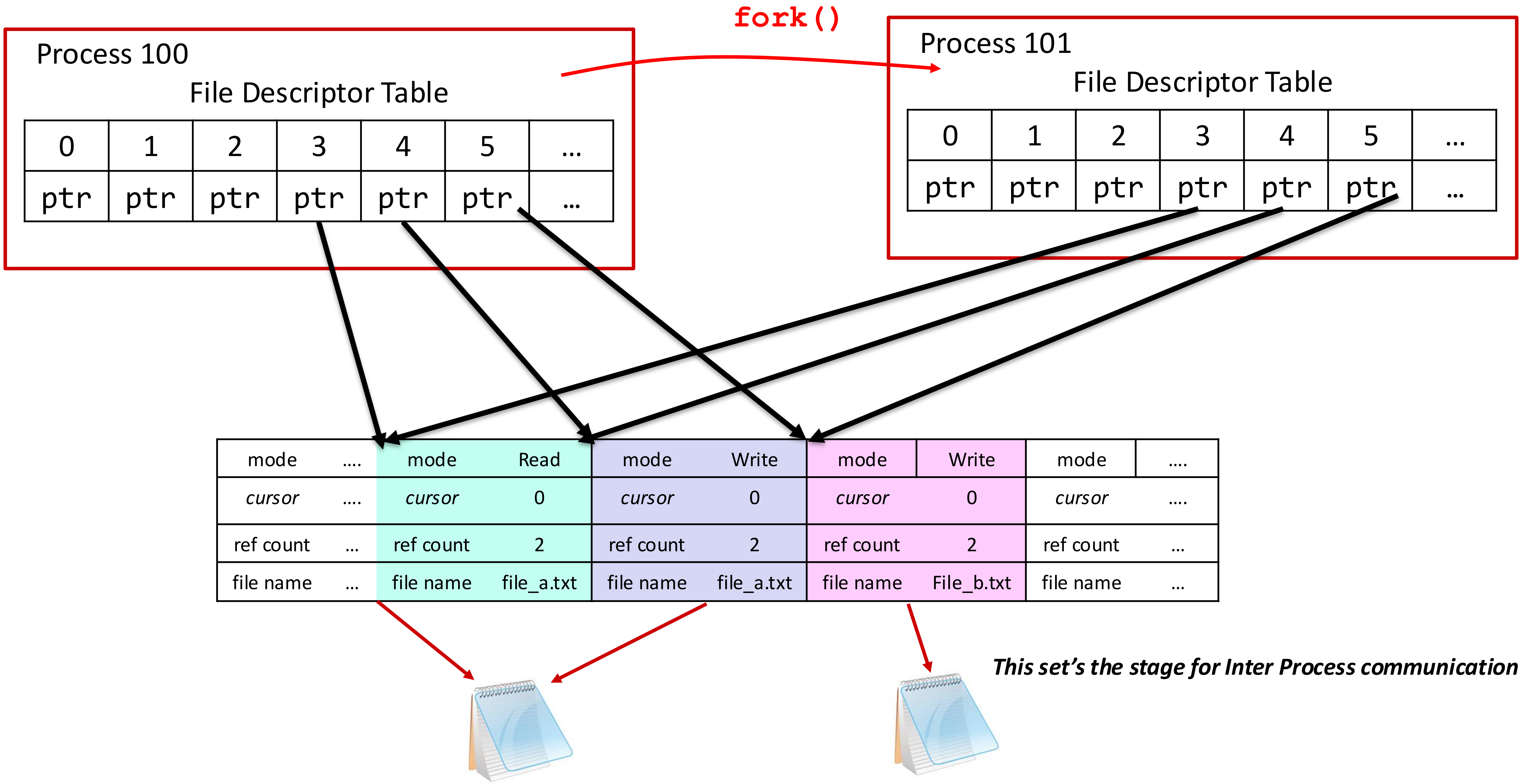
0	1	2	3	4	5	…
ptr	ptr	ptr	ptr	ptr	ptr	…

mode	….	mode	Read	mode	Write	mode	Write	mode	….
cursor	….	*cursor*	0	*cursor*	0	*cursor*	0	*cursor*	….
ref count	…	ref count	2	ref count	2	ref count	2	ref count	…
file name	…	file name	file_a.txt	file name	file_a.txt	file name	File_b.txt	file name	…

reference counts are incremented with fork!

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

The Open File Table



fork()

Process 100

File Descriptor Table

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

Process 101

File Descriptor Table

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

mode	….	mode	Read	mode	Write	mode	Write	mode	….
cursor	….	*cursor*	0	*cursor*	0	*cursor*	0	*cursor*	….
ref count	…	ref count	2	ref count	2	ref count	2	ref count	…
file name	…	file name	file_a.txt	file name	file_a.txt	file name	File_b.txt	file name	…

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

Lecture Outline

- Quick Review
  - File Descriptors
  - File Table
  - Open File Table
- Pipes and Dup
- pipe2

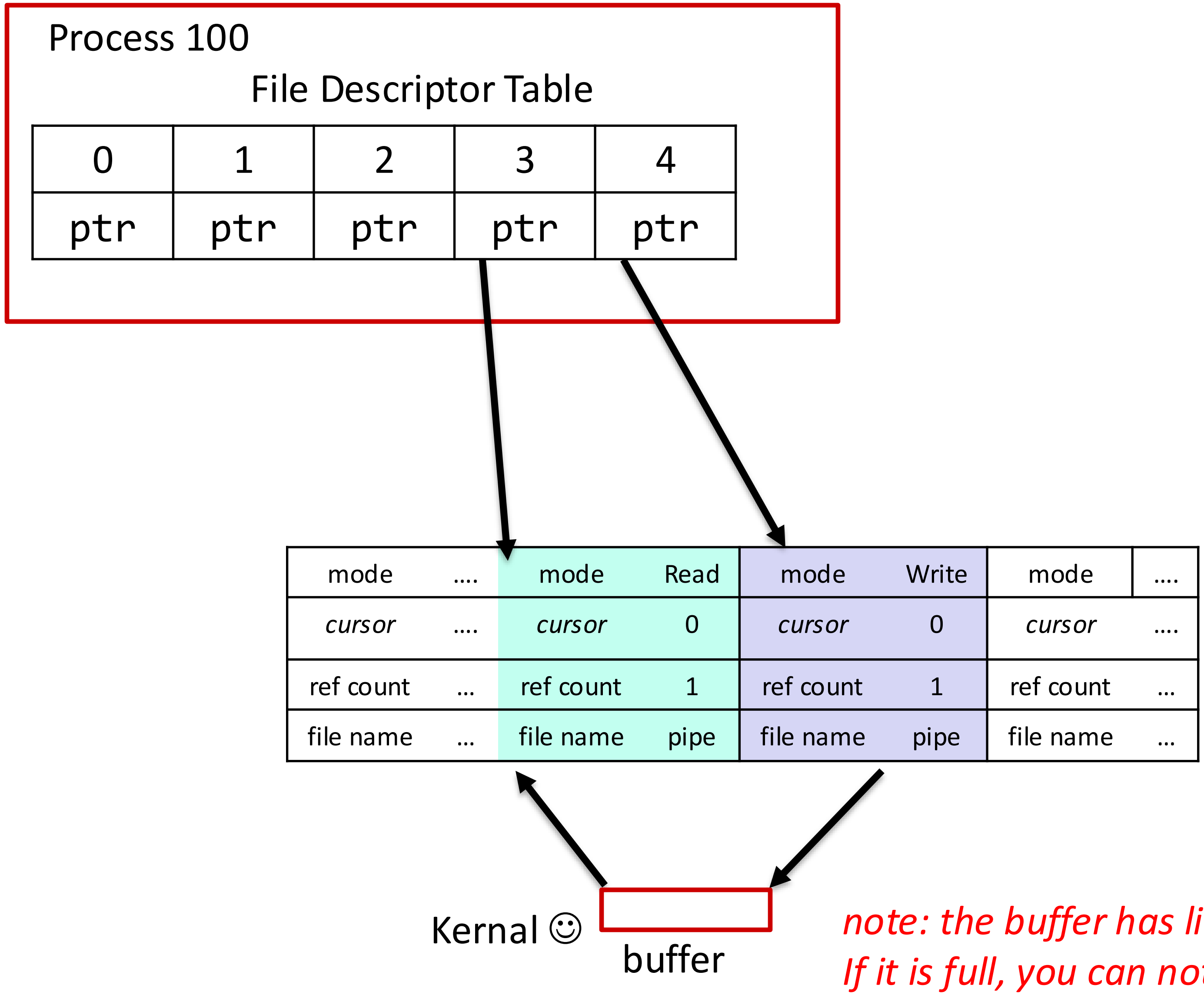
Interprocess Communication: *Pipes*

```c
int pipe(int pipefd[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

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

Visualizing Pipes



Process 100

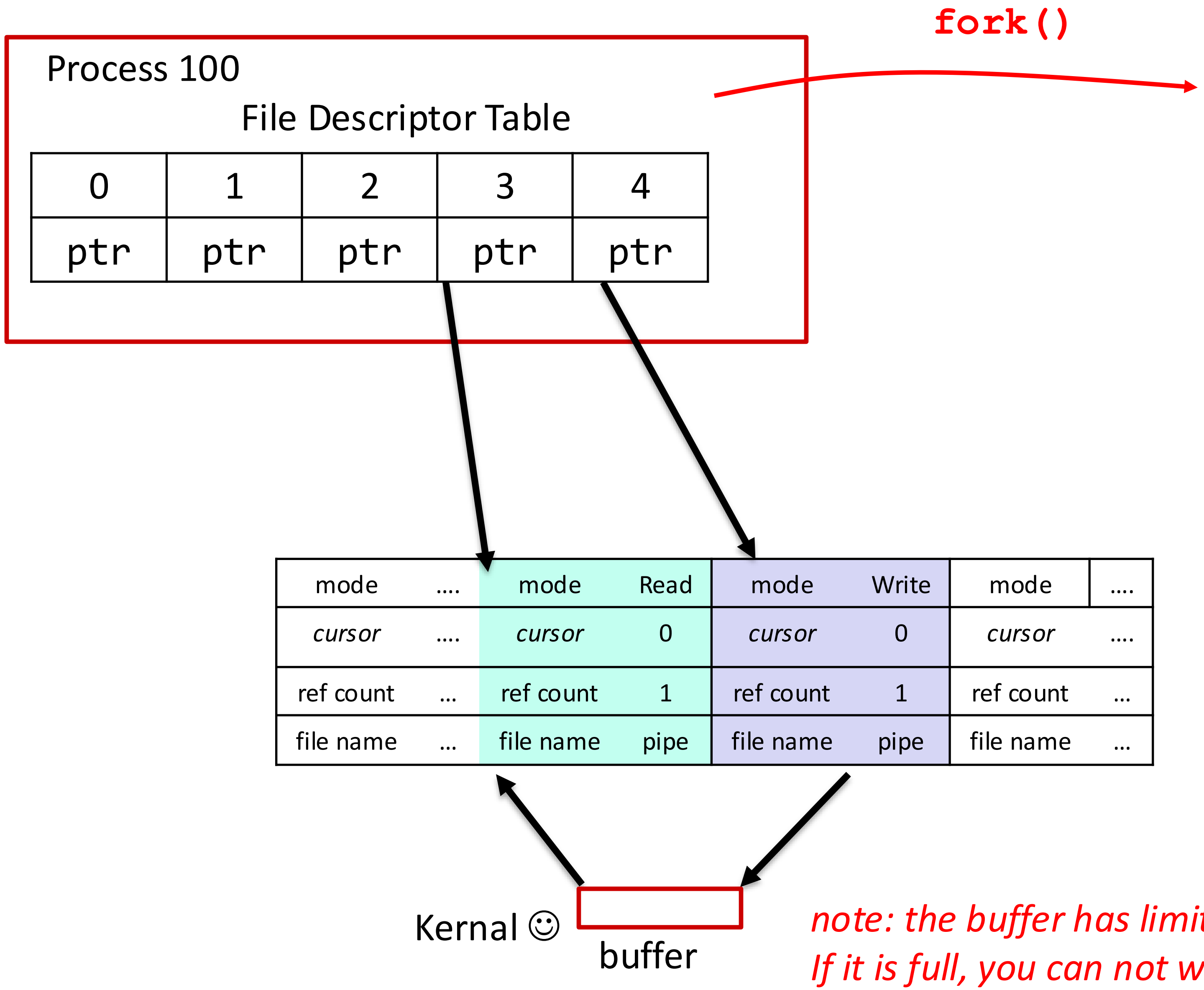
File Descriptor Table

0	1	2	3	4
`ptr`	`ptr`	`ptr`	`ptr`	`ptr`

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	...

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



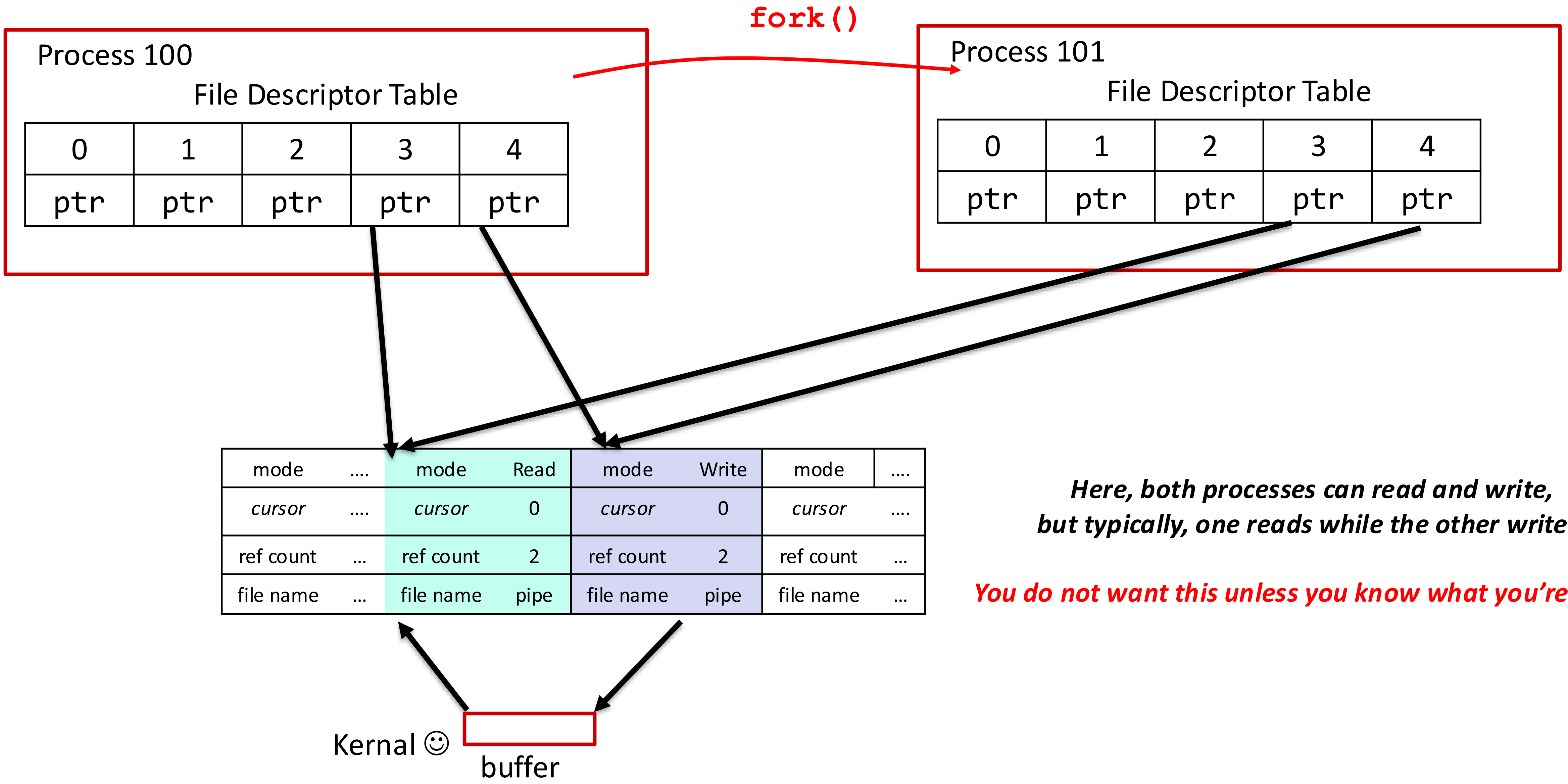
0	1	2	3	4
ptr	ptr	ptr	ptr	ptr

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	...

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



Process 100 File Descriptor Table:

0	1	2	3	4
ptr	ptr	ptr	ptr	ptr

Process 101 File Descriptor Table:

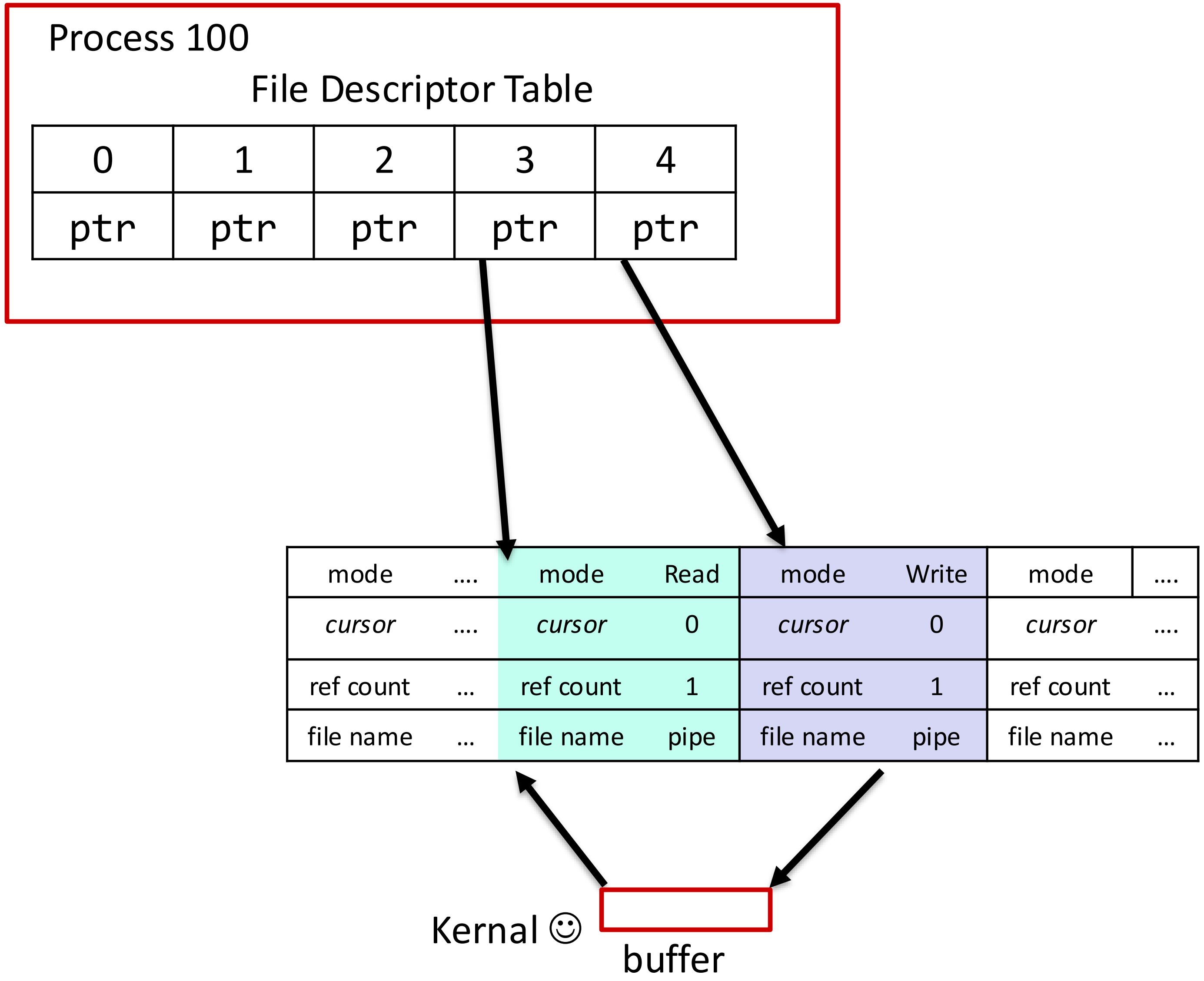
0	1	2	3	4
ptr	ptr	ptr	ptr	ptr

mode	....	mode	Read	mode	Write	mode	....
cursor	....	*cursor*	0	*cursor*	0	*cursor*	....
ref count	...	ref count	2	ref count	2	ref count	...
file name	...	file name	pipe	file name	pipe	file name	...

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.

Walk through short program



Process 100

File Descriptor Table

0	1	2	3	4
ptr	ptr	ptr	ptr	ptr

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	...

Kernal ☺

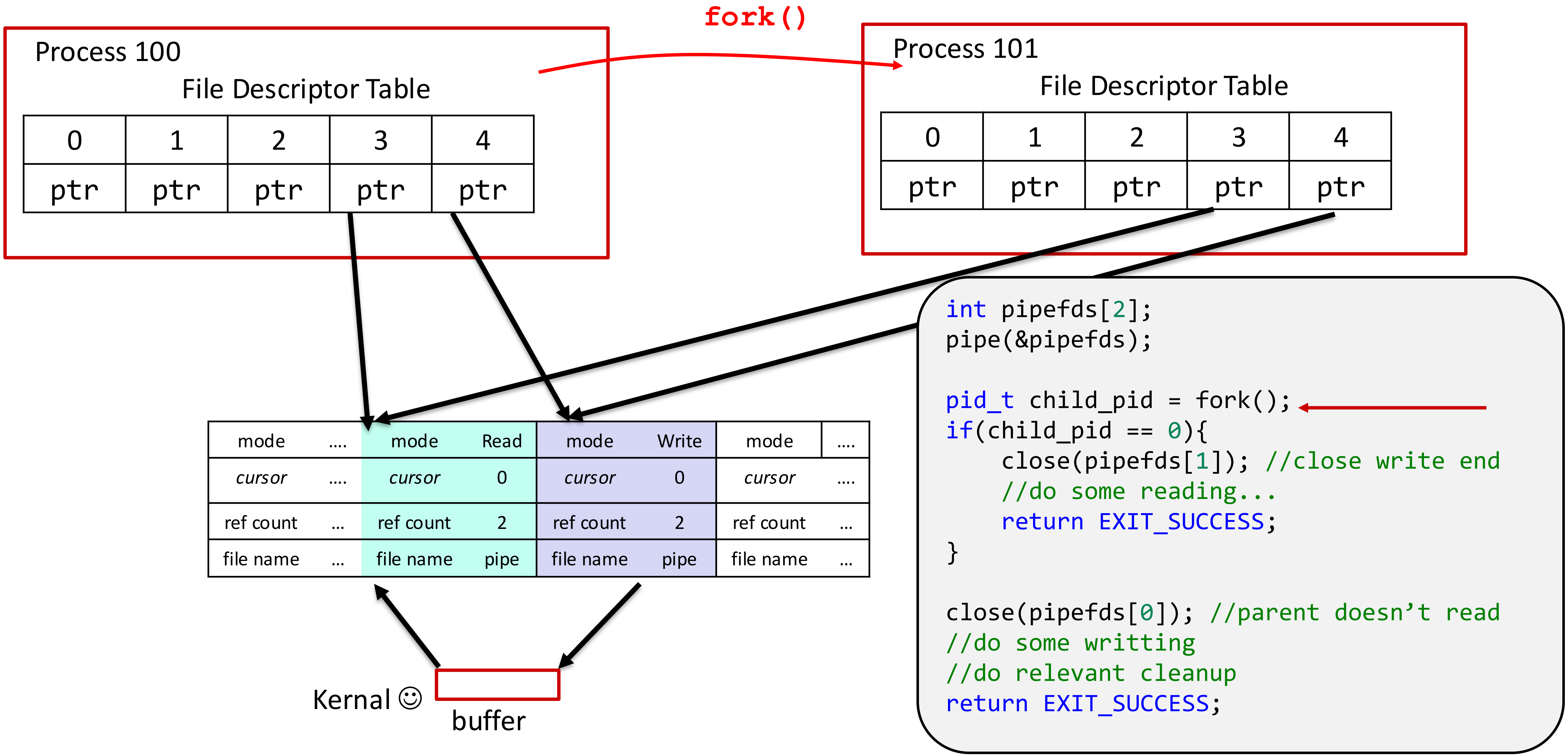
buffer

```
int pipefds[2];
pipe(&pipefds);

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

close(pipefds[0]); //parent doesn’t read
//do some writting
//do relevant cleanup
return EXIT_SUCCESS;
```

Walk through short program



Process 100

File Descriptor Table

0	1	2	3	4
ptr	ptr	ptr	ptr	ptr

Process 101

File Descriptor Table

0	1	2	3	4
ptr	ptr	ptr	ptr	ptr

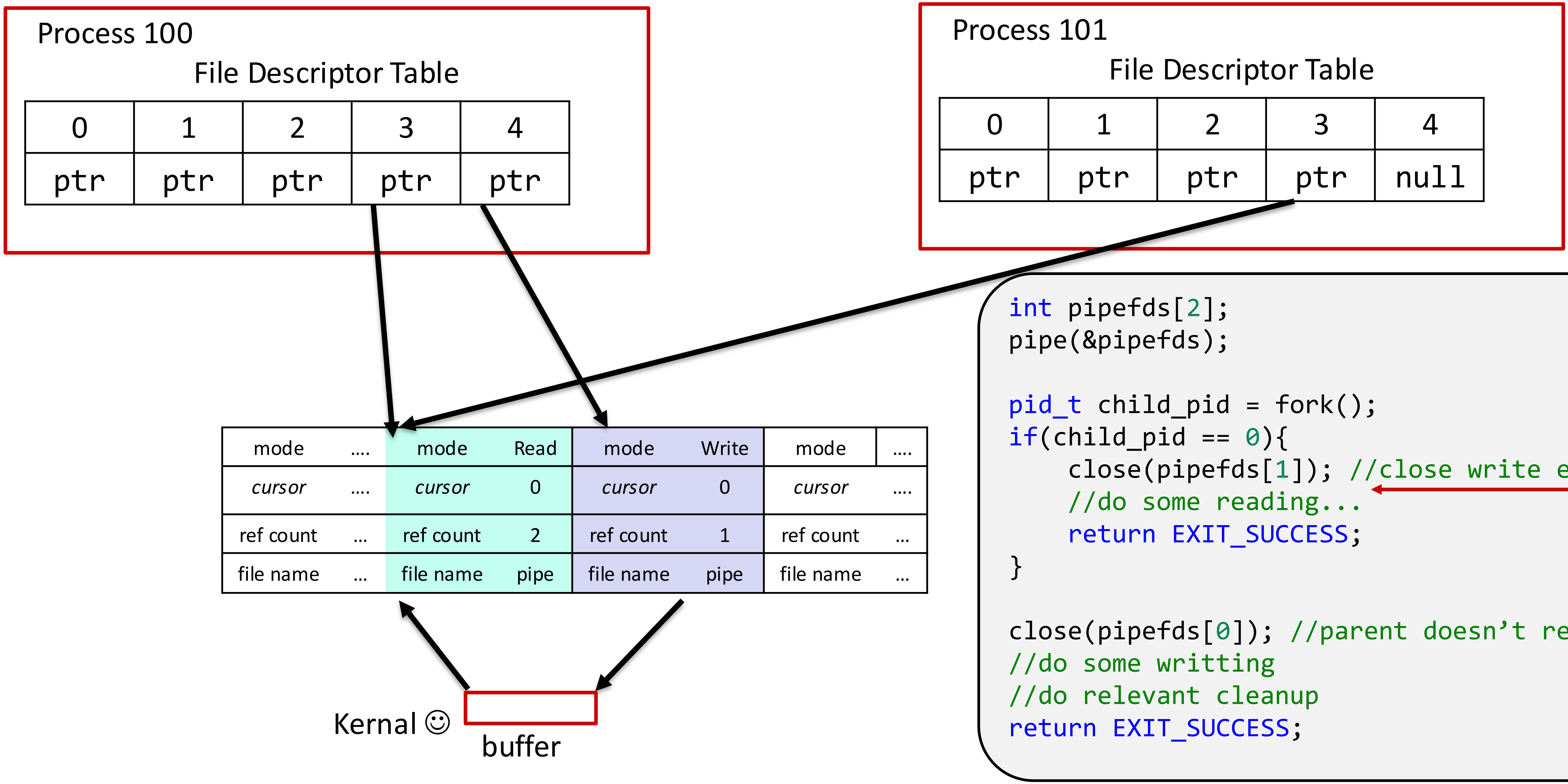
mode	....	mode	Read	mode	Write	mode	....
cursor	....	*cursor*	0	*cursor*	0	*cursor*	....
ref count	...	ref count	2	ref count	2	ref count	...
file name	...	file name	pipe	file name	pipe	file name	...

```
int pipefds[2];
pipe(&pipefds);

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

close(pipefds[0]); //parent doesn’t read
//do some writting
//do relevant cleanup
return EXIT_SUCCESS;
```

Walk through short program



Process 100

File Descriptor Table

0	1	2	3	4
ptr	ptr	ptr	ptr	ptr

Process 101

File Descriptor Table

0	1	2	3	4
ptr	ptr	ptr	ptr	null

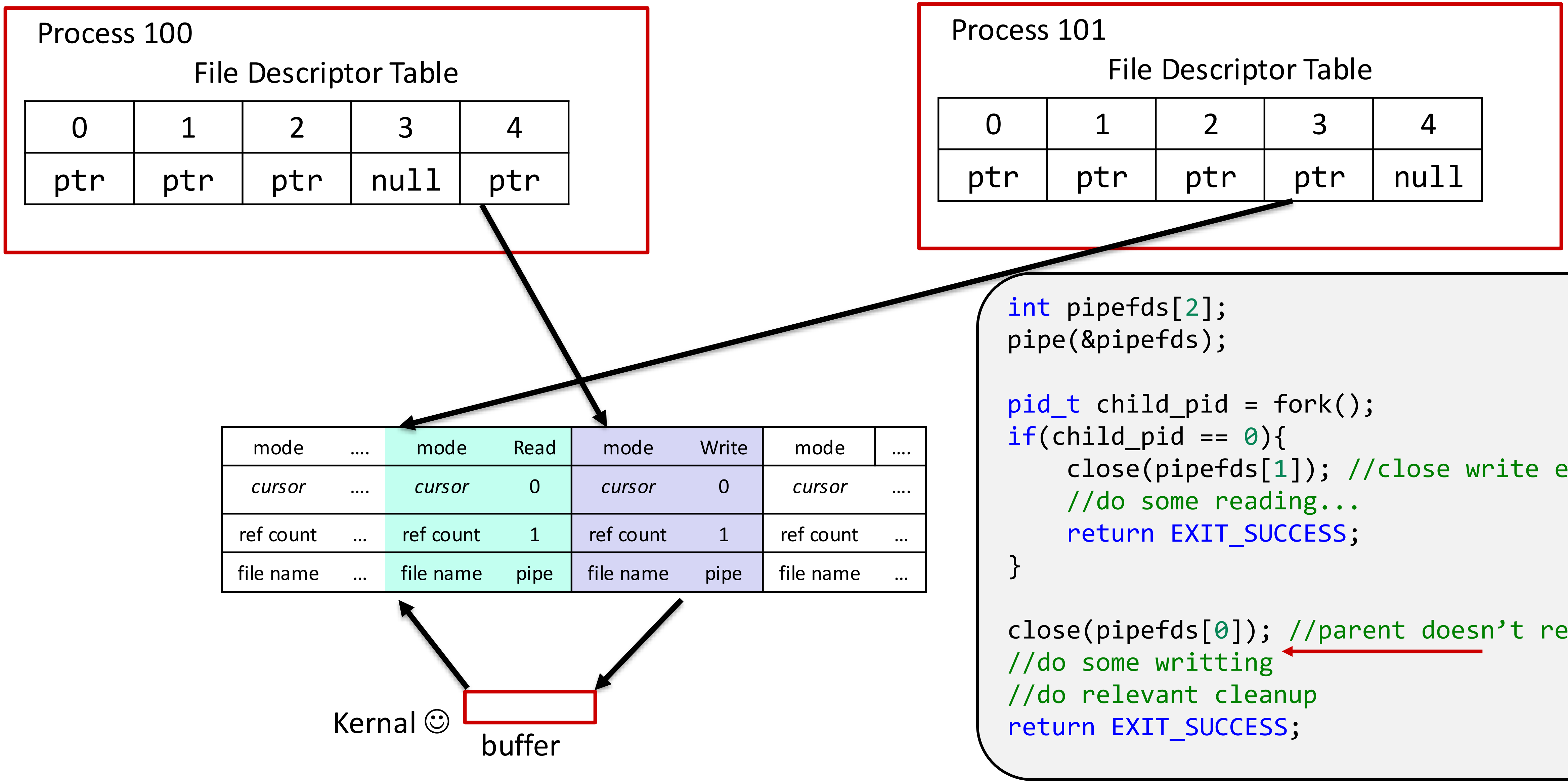
mode	....	mode	Read	mode	Write	mode	....
cursor	....	*cursor*	0	*cursor*	0	*cursor*	....
ref count	...	ref count	2	ref count	1	ref count	...
file name	...	file name	pipe	file name	pipe	file name	...

```
int pipefds[2];
pipe(&pipefds);

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

close(pipefds[0]); //parent doesn't read
//do some writting
//do relevant cleanup
return EXIT_SUCCESS;
```

Walk through short program



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	...

Kernal ☺

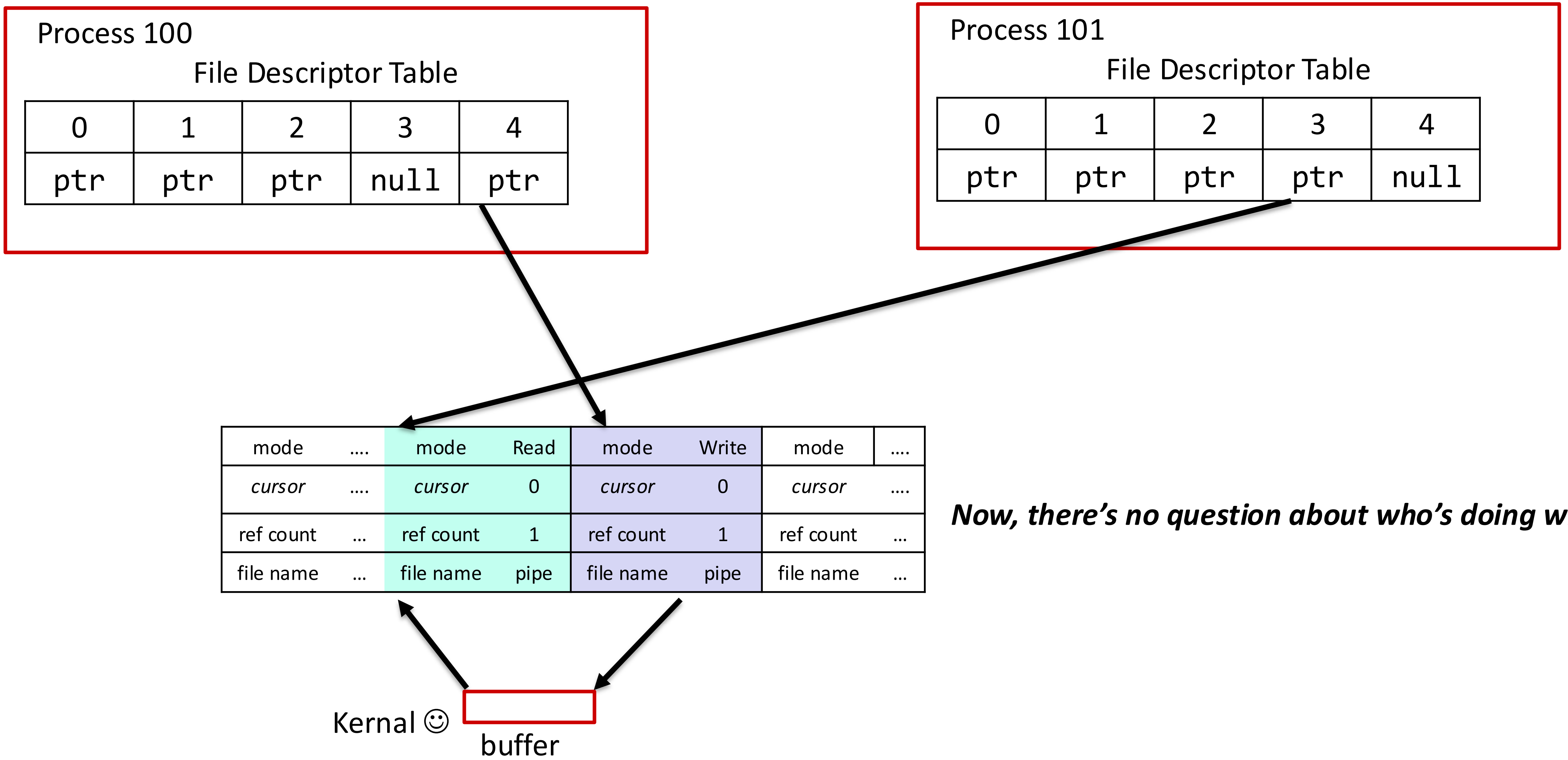
buffer

```
int pipefds[2];
pipe(&pipefds);

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

close(pipefds[0]); //parent doesn’t read
//do some writting
//do relevant cleanup
return EXIT_SUCCESS;
```

Final State of Short Program



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	...

Kernal ☺

buffer

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

dup2: redirecting to our heart's desire

- 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-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`"

Piping in the Shell

```
cat bee_movie.txt | grep Barry | uniq
```

- ***cat*** first outputs the entire contents of bee_movie.txt and pipes it into ***grep***, which filters for lines containing "Barry"
- 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.

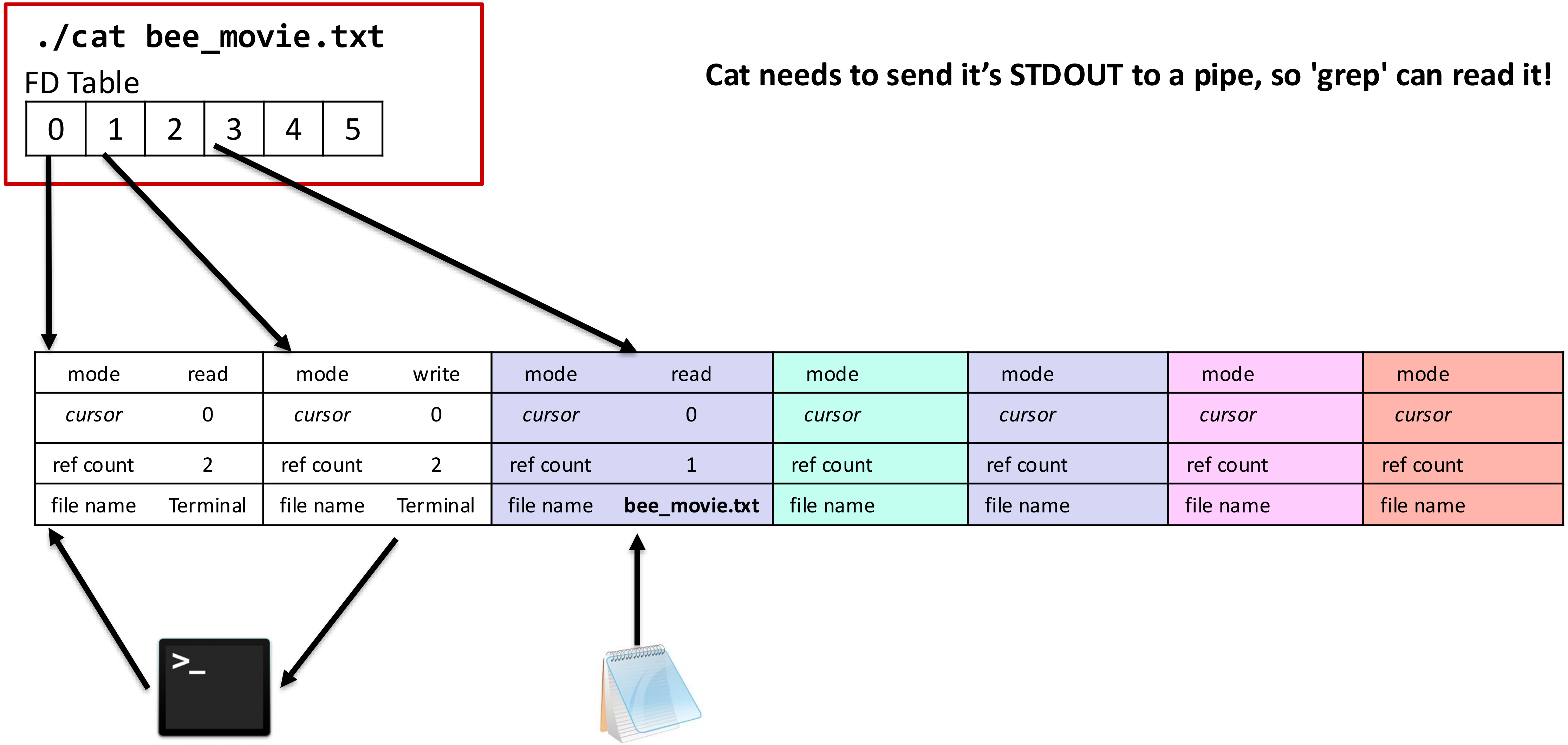
Poll Everywhere

pollev.com/cis5480

```
cat bee_movie.txt | grep Barry | uniq
```

How many pipes do we need to execute this command?

`cat bee_movie.txt | grep Barry | uniq`



`./cat bee_movie.txt`

FD Table

0	1	2	3	4	5

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

mode	read	mode	write	mode	read	mode	mode	mode	mode
cursor	0	*cursor*	0	*cursor*	0	*cursor*	*cursor*	*cursor*	*cursor*
ref count	2	ref count	2	ref count	1	ref count	ref count	ref count	ref count
file name	Terminal	file name	Terminal	file name	**bee_movie.txt**	file name	file name	file name	file name

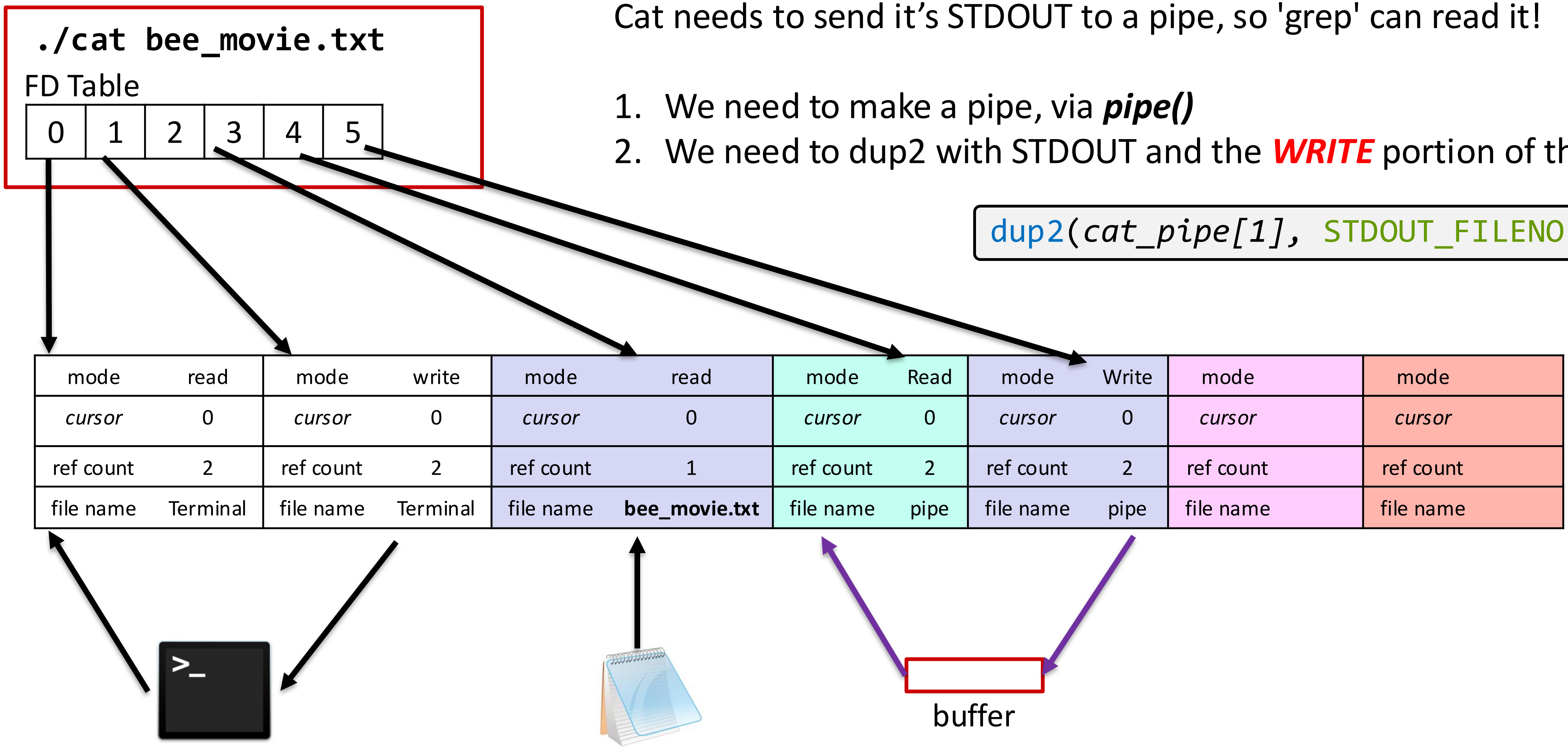
Note: the ref counts might seem inflated, but there is a shell process that exists too and forks these processes.

`cat bee_movie.txt | grep Barry | 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 bee_movie.txt`

FD Table

0	1	2	3	4	5

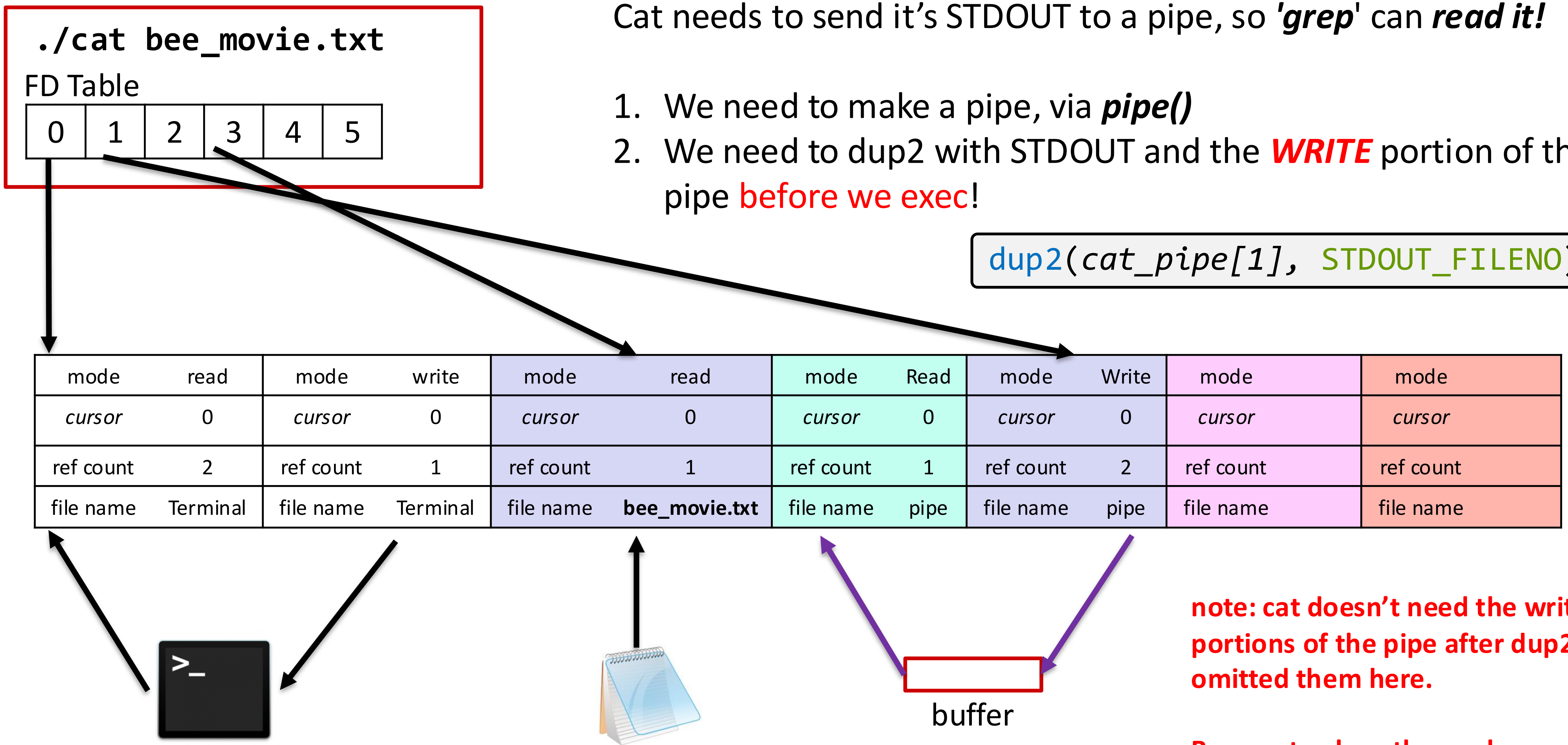
mode	read	mode	write	mode	read	mode	Read	mode	Write	mode		mode	
cursor	0	*cursor*	0	*cursor*	0	*cursor*	0	*cursor*	0	*cursor*		*cursor*	
ref count	2	ref count	2	ref count	1	ref count	2	ref count	2	ref count		ref count	
file name	Terminal	file name	Terminal	file name	**bee_movie.txt**	file name	pipe	file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

cat bee_movie.txt | grep Barry | 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);
```



```
./cat bee_movie.txt
```

FD Table

0	1	2	3	4	5

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	**bee_movie.txt**	file name	pipe	file name	pipe	file name		file name	

note: cat doesn’t need the write or read portions of the pipe after dup2, so I’ve omitted them here.

Be sure to close them when not necessary. We’ll see a better trick in a bit.

Poll Everywhere

pollev.com/cis5480

`cat bee_movie.txt | grep Barry | uniq`

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

```c
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/cis5480

cat bee_movie.txt | grep Barry | uniq

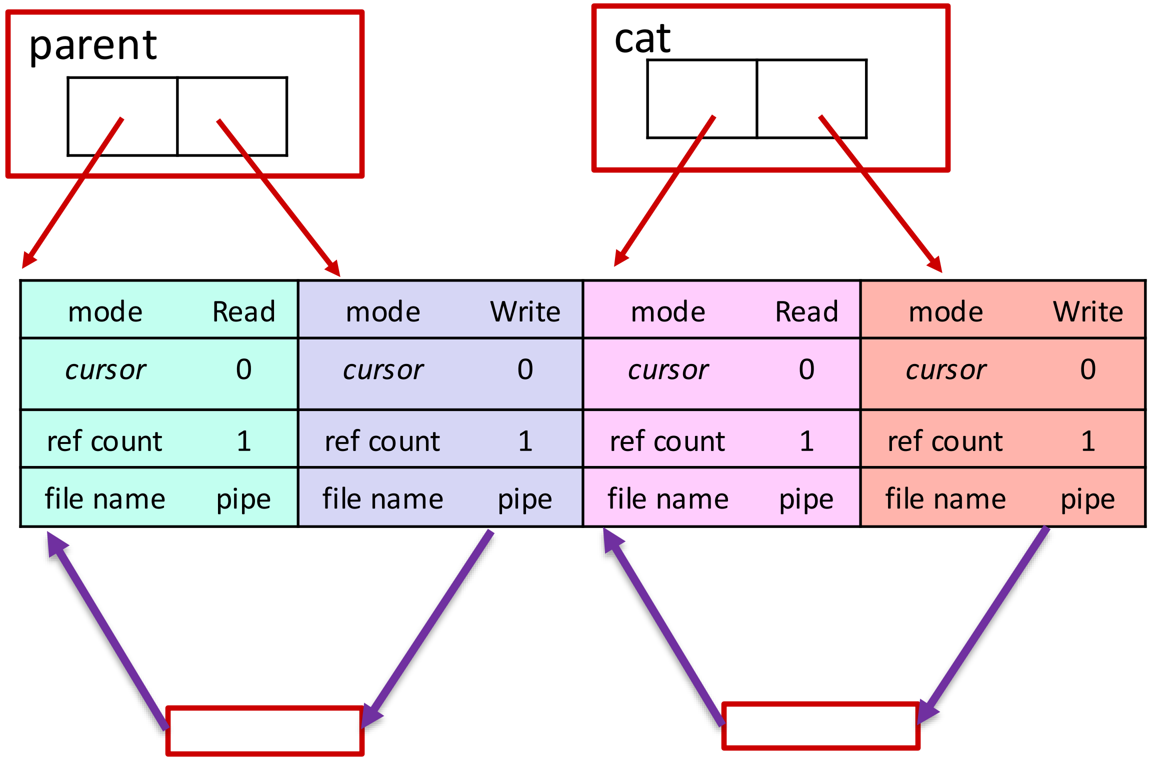
Where can we put a pipe, so both cat and grep can write and read, respectively?

```
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 FD are NOT SHARED!***



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

Poll Everywhere

pollev.com/cis5480

`cat bee_movie.txt | grep Barry | 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 it's 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/cis5480

`cat bee_movie.txt | grep Barry | uniq`

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

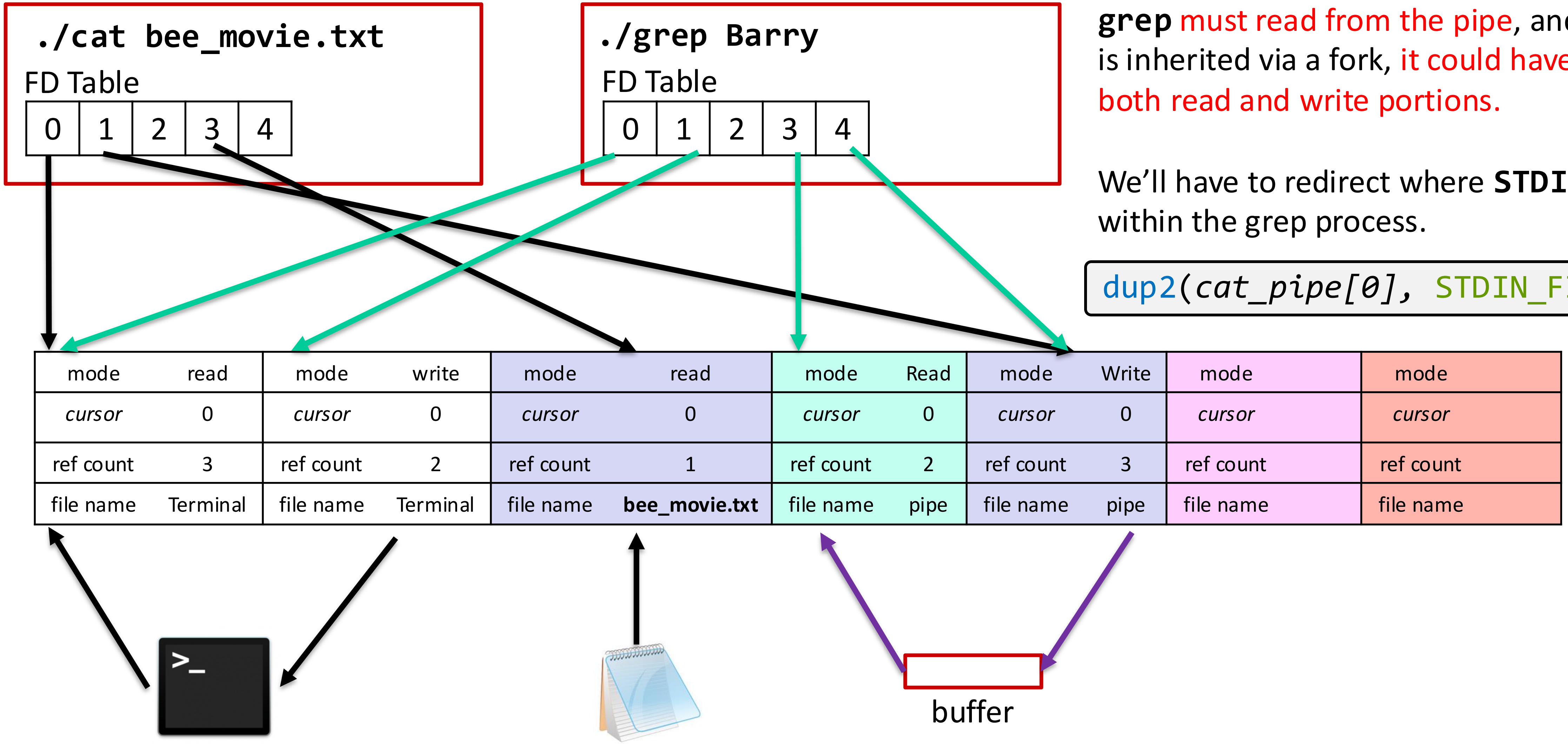
```
int cat_pipe[2];

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 sepearte pipe in the parent and the grep process. No way to wrangle the pipes this way.***

`cat bee_movie.txt | grep Barry | uniq`



./cat bee_movie.txt

FD Table

0	1	2	3	4

./grep Barry

FD Table

0	1	2	3	4

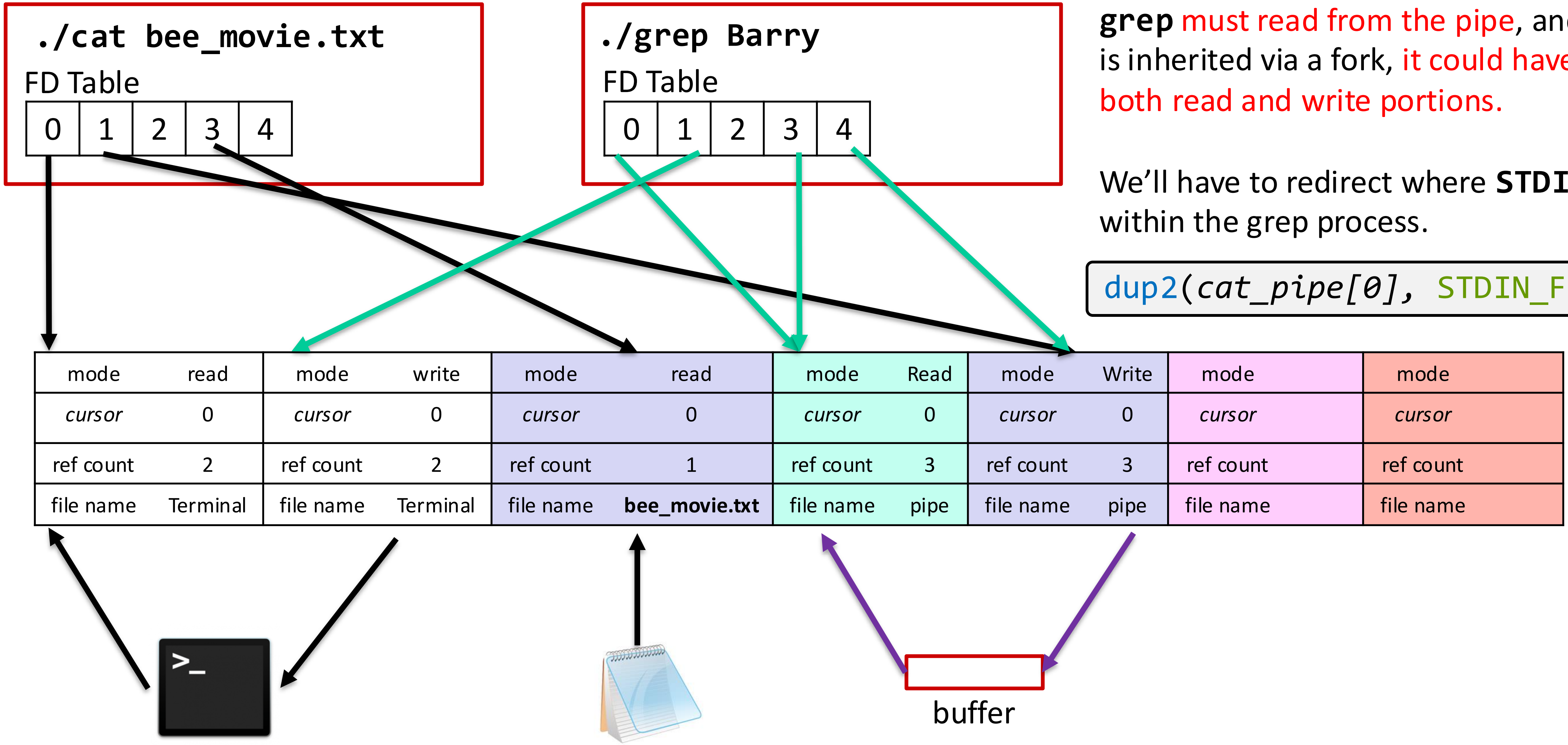
mode	read	mode	write	mode	read	mode	Read	mode	Write	mode		mode	
cursor	0	*cursor*	0	*cursor*	0	*cursor*	0	*cursor*	0	*cursor*		*cursor*	
ref count	3	ref count	2	ref count	1	ref count	2	ref count	3	ref count		ref count	
file name	Terminal	file name	Terminal	file name	**bee_movie.txt**	file name	pipe	file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

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 bee_movie.txt | grep Barry | uniq`



`./cat bee_movie.txt`

FD Table

0	1	2	3	4

`./grep Barry`

FD Table

0	1	2	3	4

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	3	ref count	3	ref count		ref count	
file name	Terminal	file name	Terminal	file name	**bee_movie.txt**	file name	pipe	file name	pipe	file name		file name	

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.

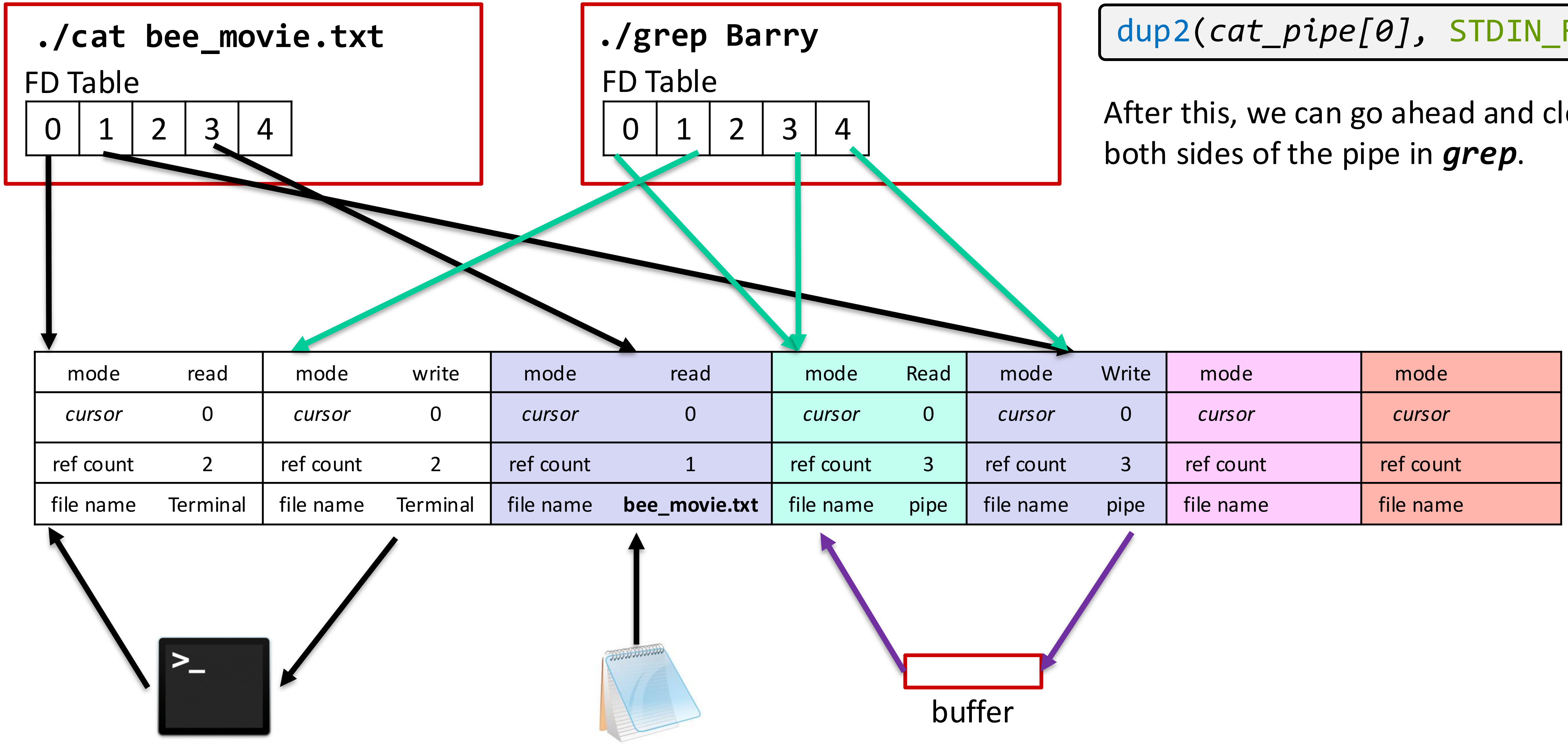
We'll have to redirect where **STDIN** refers to within the grep process.

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

`cat bee_movie.txt | grep Barry | uniq`

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

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



`./cat bee_movie.txt`

FD Table

0	1	2	3	4

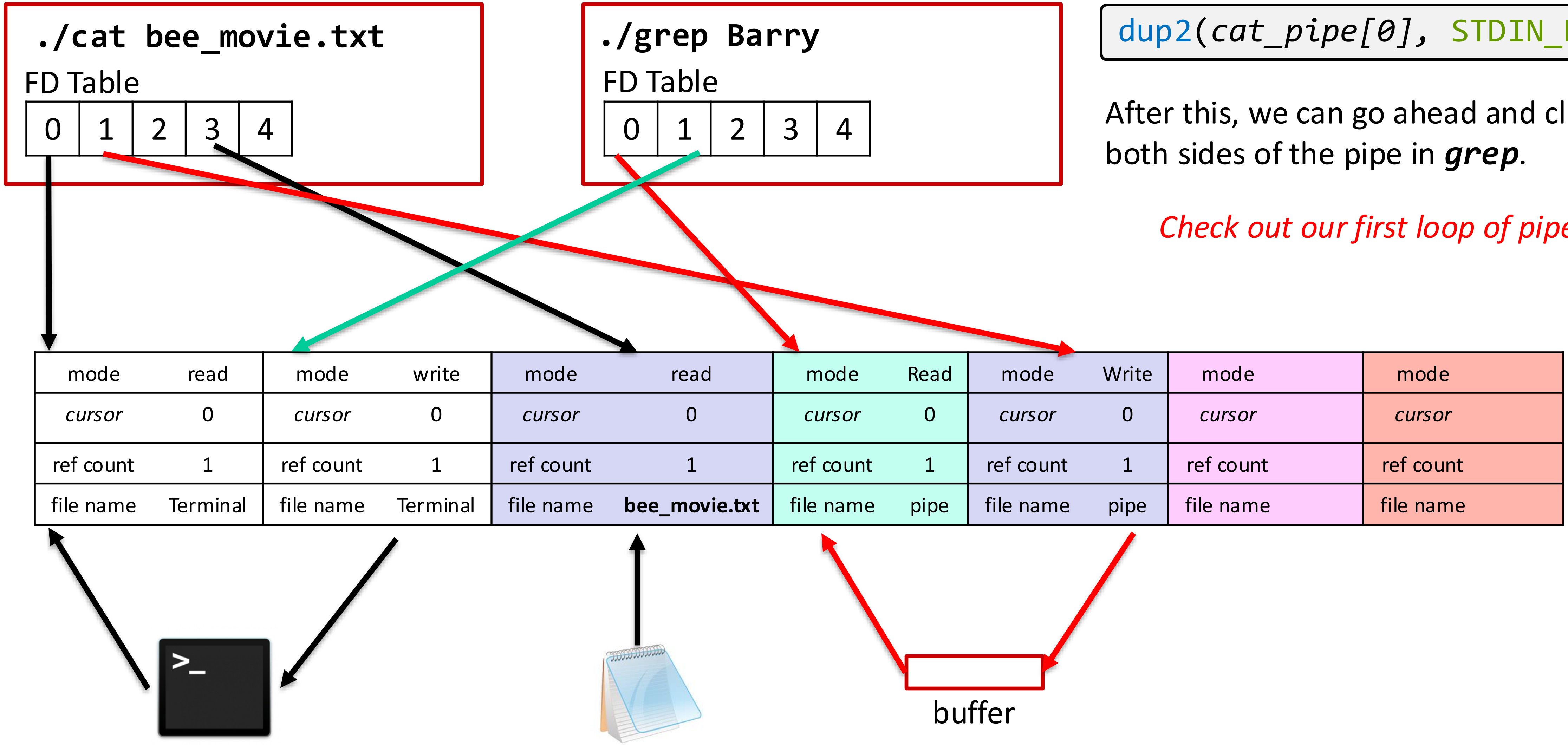
`./grep Barry`

FD Table

0	1	2	3	4

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	3	ref count	3	ref count		ref count	
file name	Terminal	file name	Terminal	file name	**bee_movie.txt**	file name	pipe	file name	pipe	file name		file name	

`cat bee_movie.txt | grep Barry | uniq`



./cat bee_movie.txt

FD Table

0	1	2	3	4

./grep Barry

FD Table

0	1	2	3	4

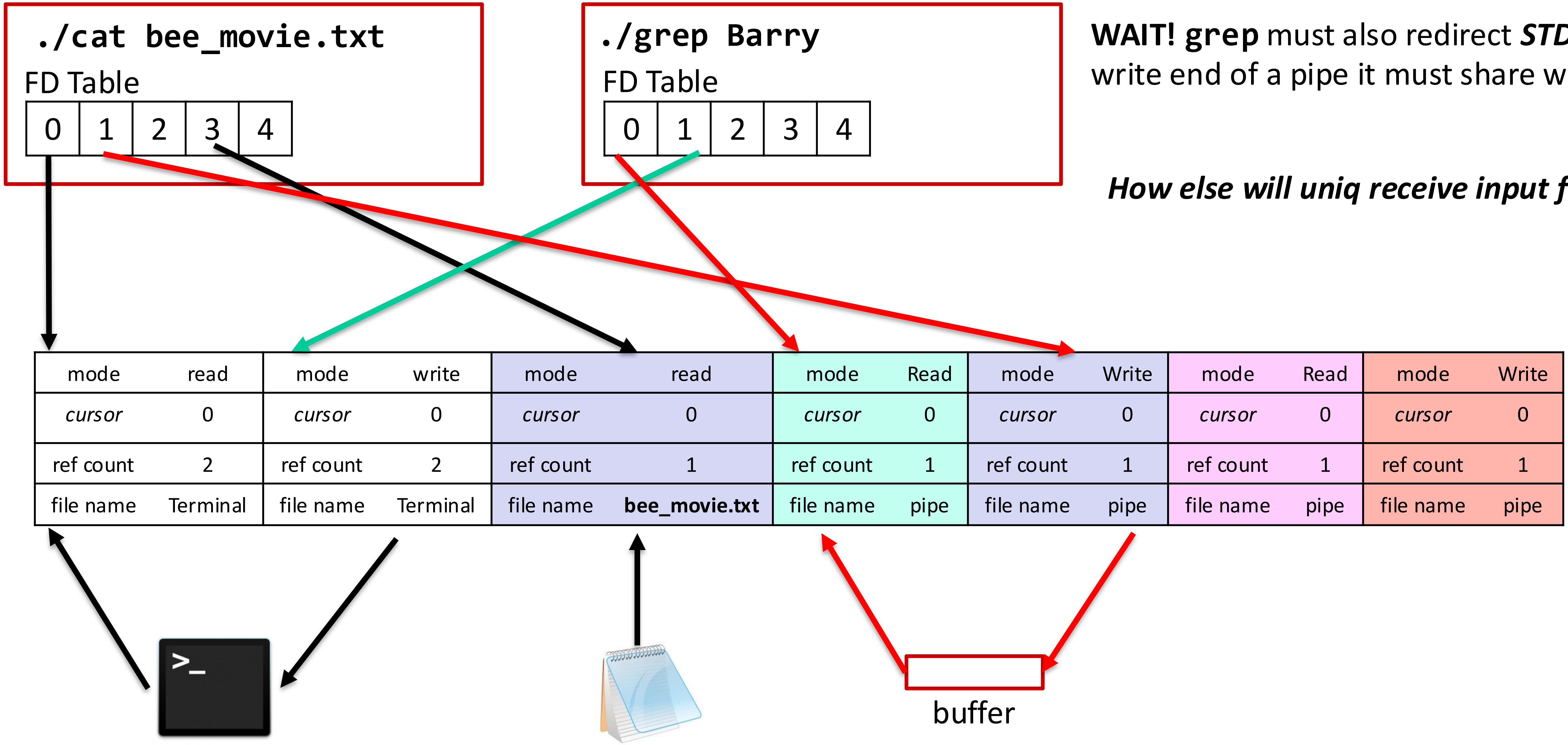
mode	read	mode	write	mode	read	mode	Read	mode	Write	mode		mode	
cursor	0	*cursor*	0	*cursor*	0	*cursor*	0	*cursor*	0	*cursor*		*cursor*	
ref count	1	ref count	1	ref count	1	ref count	1	ref count	1	ref count		ref count	
file name	Terminal	file name	Terminal	file name	**bee_movie.txt**	file name	pipe	file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

```
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!

`cat bee_movie.txt | grep Barry | uniq`

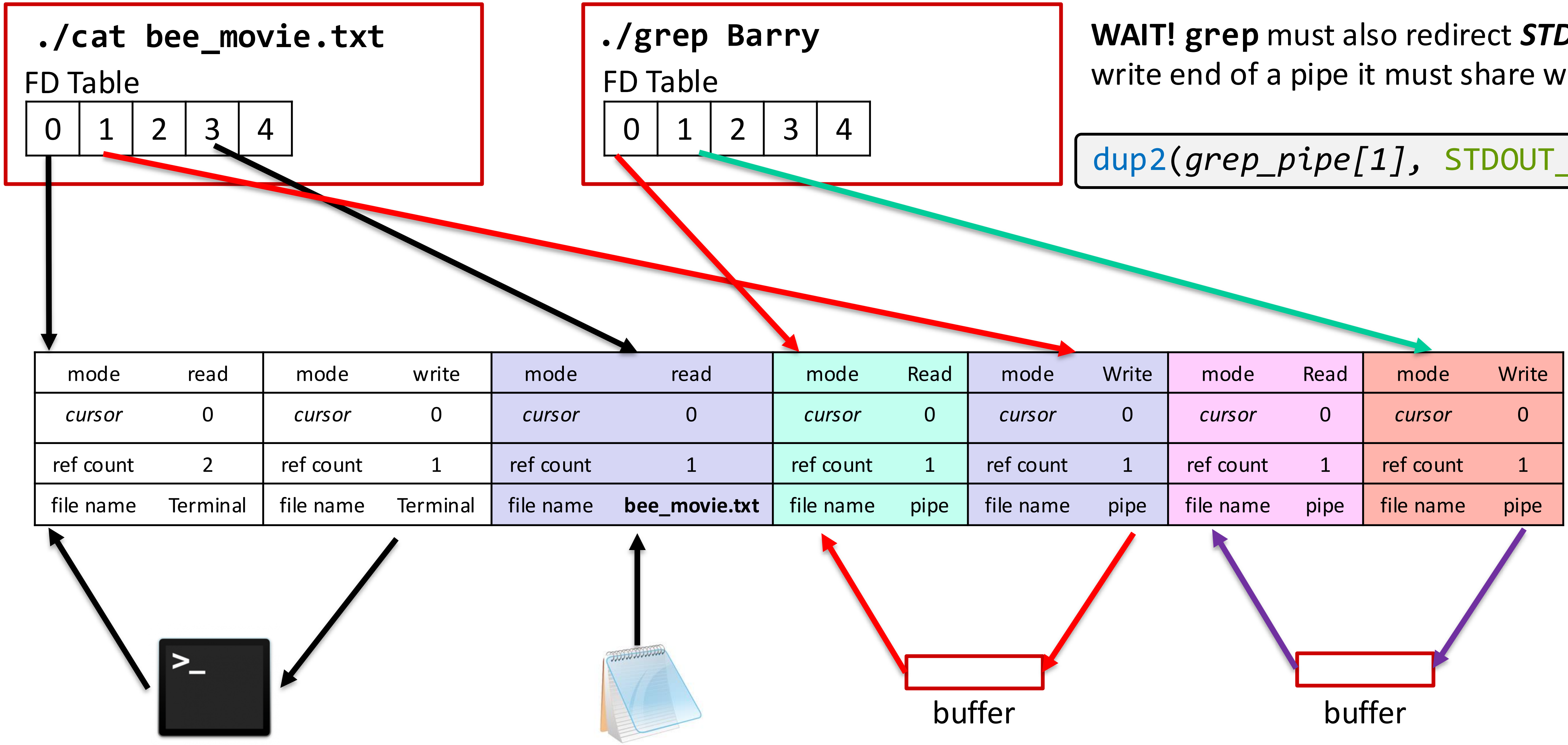


mode	read	mode	write	mode	read	mode	Read	mode	Write	mode	Read	mode	Write
cursor	0	*cursor*	0	*cursor*	0	*cursor*	0	*cursor*	0	*cursor*	0	*cursor*	0
ref count	2	ref count	2	ref count	1	ref count	1	ref count	1	ref count	1	ref count	1
file name	Terminal	file name	Terminal	file name	**bee_movie.txt**	file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

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?

cat bee_movie.txt | grep Barry | uniq



./cat bee_movie.txt

FD Table

0	1	2	3	4

./grep Barry

FD Table

0	1	2	3	4

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	**bee_movie.txt**	file name	pipe	file name	pipe	file name	pipe	file name	pipe

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

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

Poll Everywhere

pollev.com/cis5480

```
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 bee_movie.txt | grep Barry | 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/cis5480

```
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 bee_movie.txt | grep Barry | 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/cis5480

```
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 bee_movie.txt | grep Barry | 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

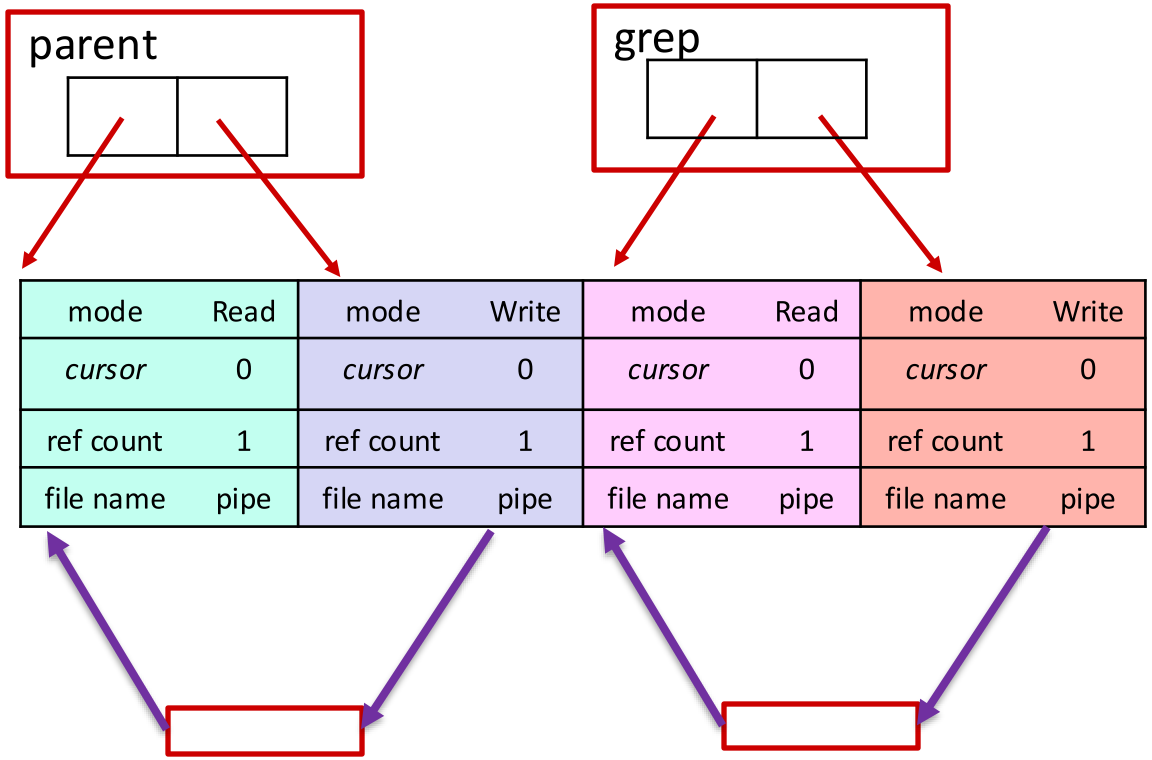
pollev.com/cis5480

```
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 bee_movie.txt | grep Barry | 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*?**



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

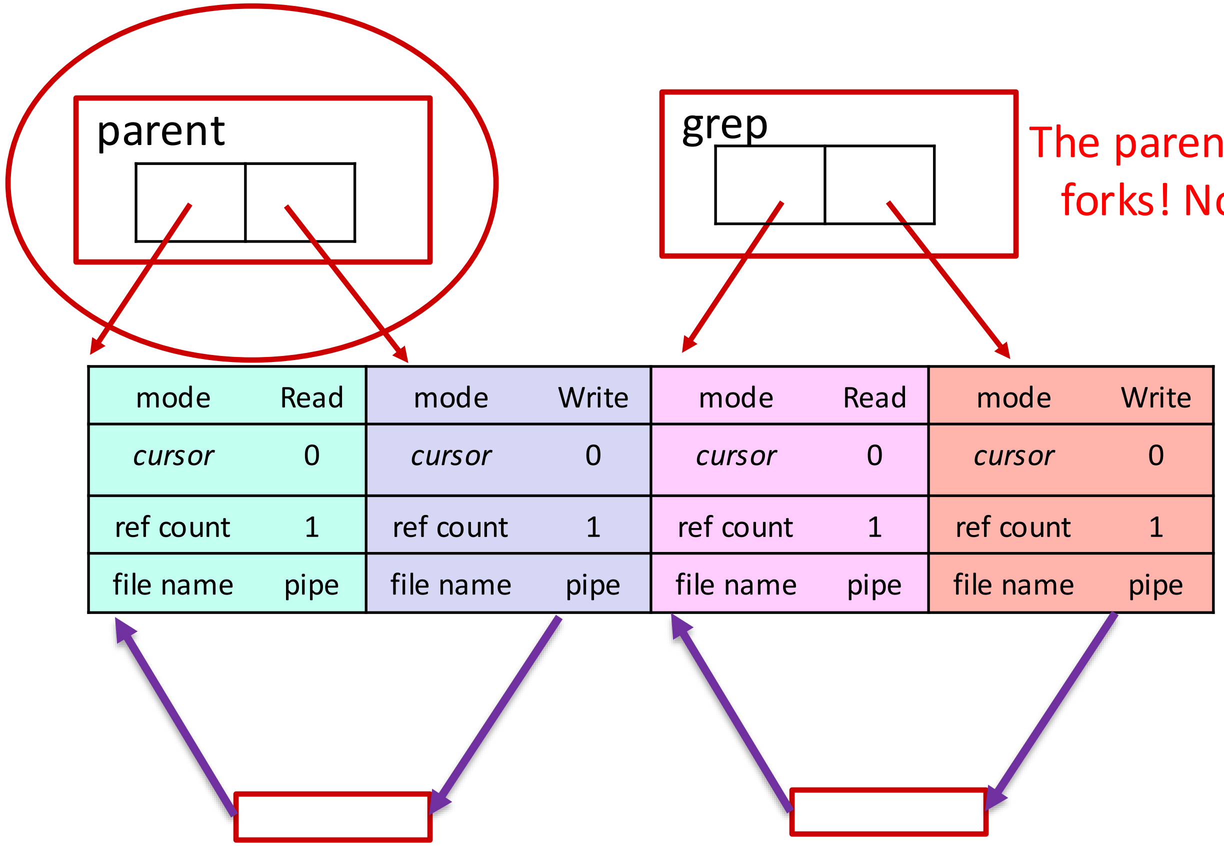
Poll Everywhere

pollev.com/cis5480

```
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 bee_movie.txt | grep Barry | 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***?

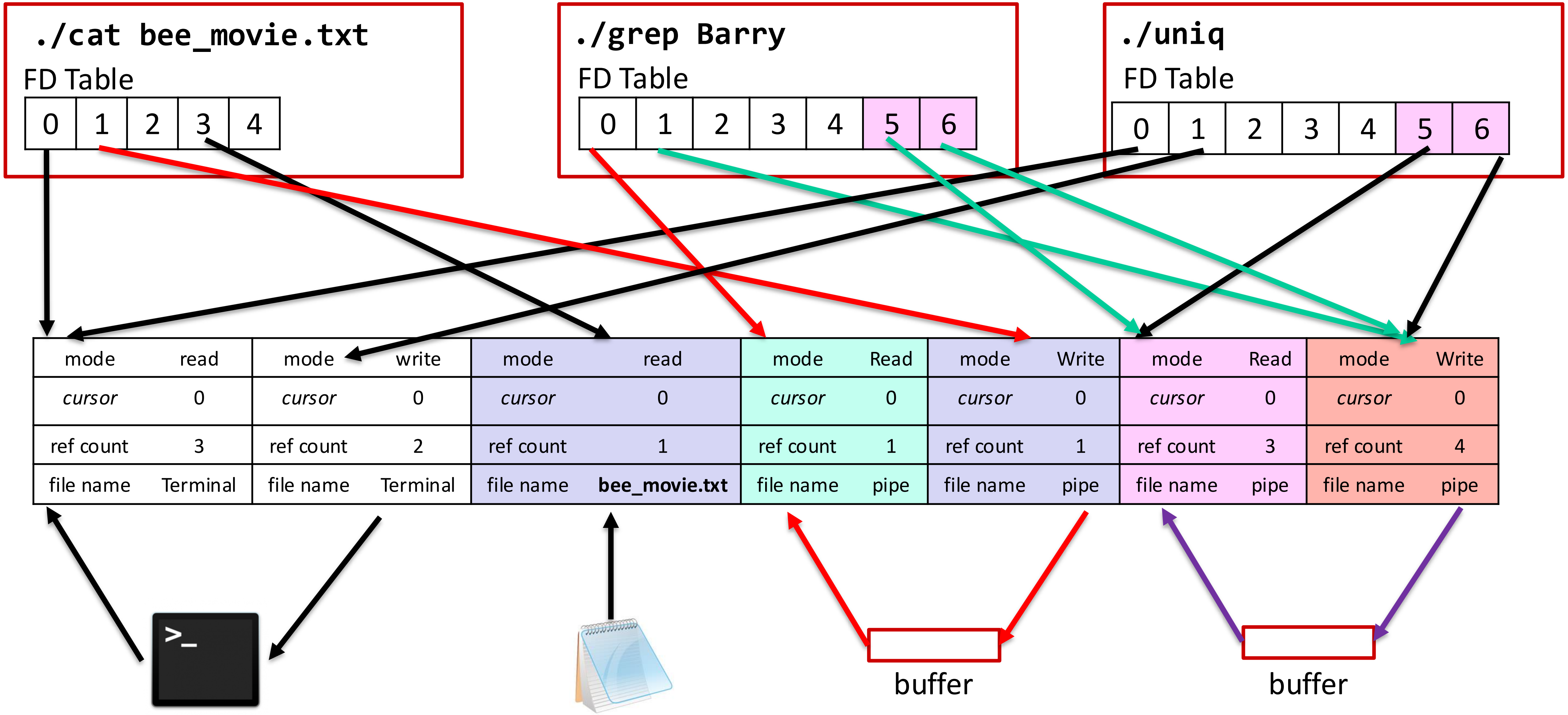


mode	Read	mode	Write	mode	Read	mode	Write
cursor	0	*cursor*	0	*cursor*	0	*cursor*	0
ref count	1	ref count	1	ref count	1	ref count	1
file name	pipe	file name	pipe	file name	pipe	file name	pipe

The parent process forks! Not grep!

cat bee_movie.txt | grep Barry | uniq

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



./cat bee_movie.txt

FD Table

0	1	2	3	4

./grep Barry

FD Table

0	1	2	3	4	5	6

./uniq

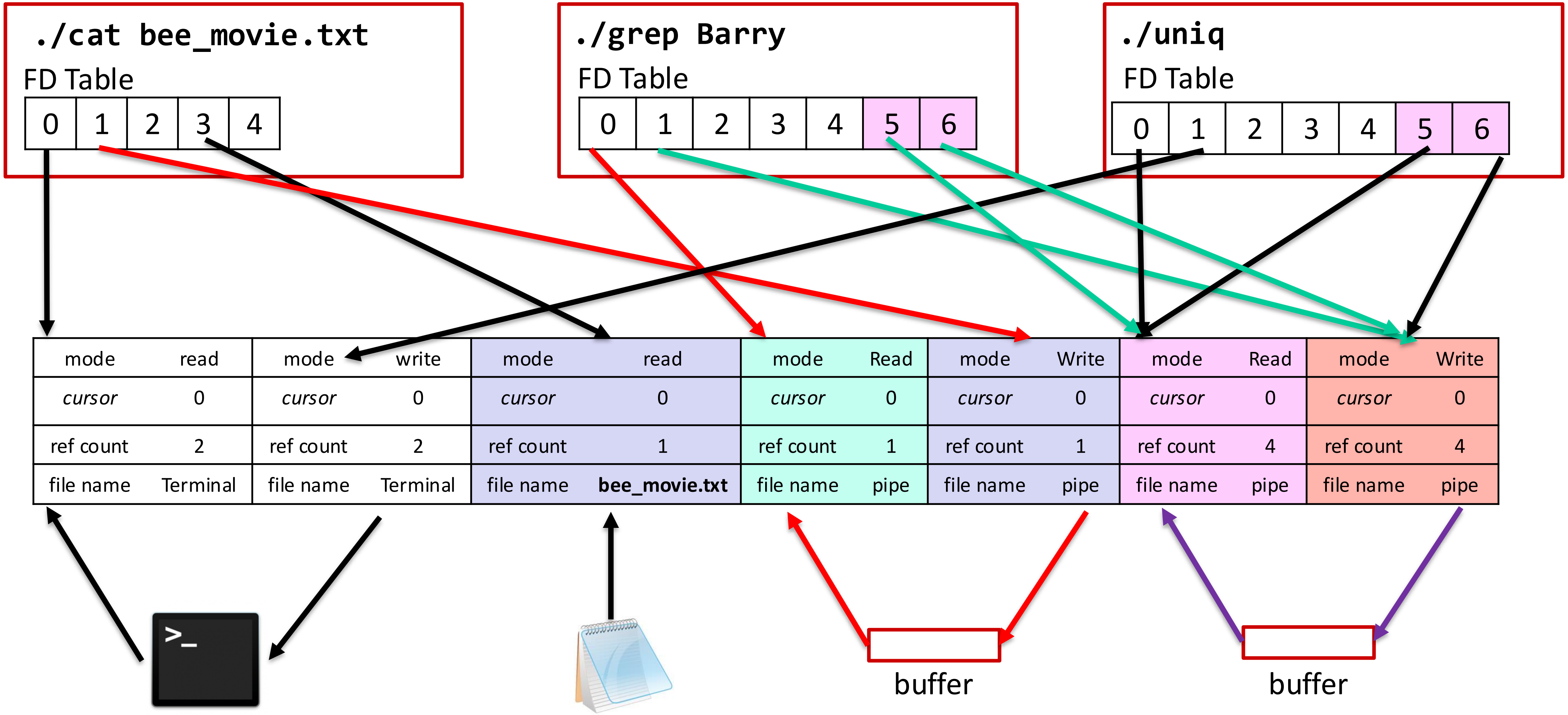
FD Table

0	1	2	3	4	5	6

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	3	ref count	2	ref count	1	ref count	1	ref count	1	ref count	3	ref count	4
file name	Terminal	file name	Terminal	file name	**bee_movie.txt**	file name	pipe	file name	pipe	file name	pipe	file name	pipe

`cat bee_movie.txt | grep Barry | uniq`

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



`./cat bee_movie.txt`

FD Table

0	1	2	3	4

`./grep Barry`

FD Table

0	1	2	3	4	5	6

`./uniq`

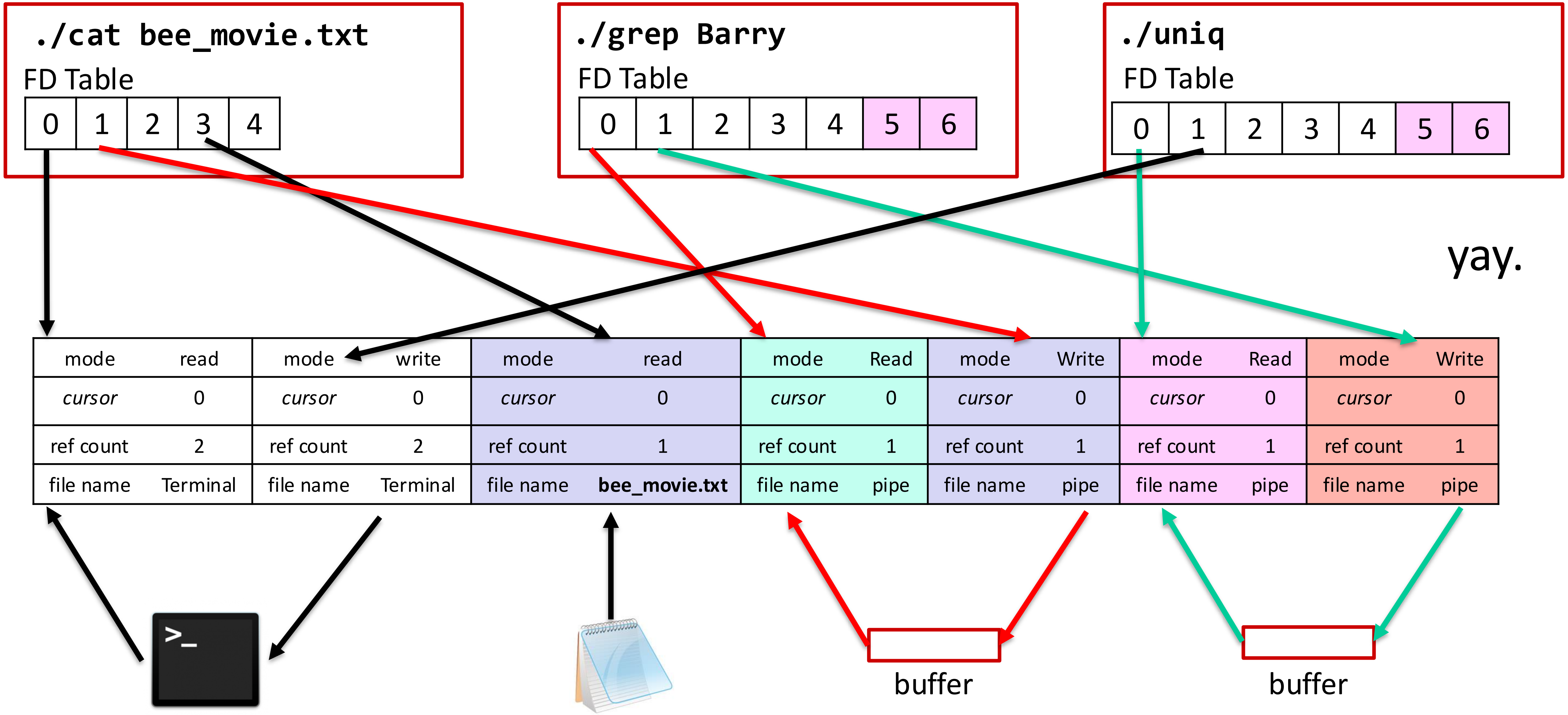
FD Table

0	1	2	3	4	5	6

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	4	ref count	4
file name	Terminal	file name	Terminal	file name	**bee_movie.txt**	file name	pipe	file name	pipe	file name	pipe	file name	pipe

cat bee_movie.txt | grep Barry | uniq

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



`./cat bee_movie.txt`

FD Table

0	1	2	3	4

`./grep Barry`

FD Table

0	1	2	3	4	5	6

`./uniq`

FD Table

0	1	2	3	4	5	6

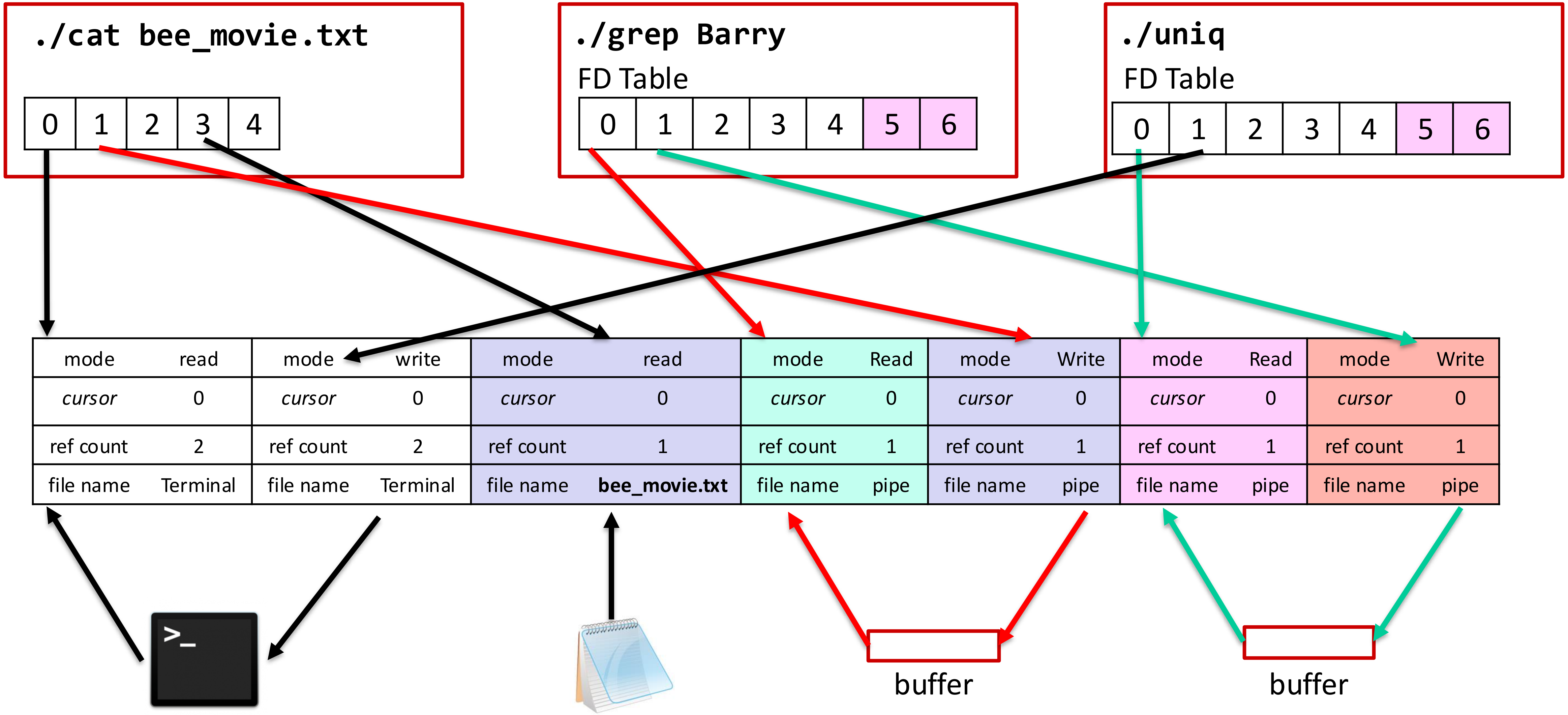
yay.

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	**bee_movie.txt**	file name	pipe	file name	pipe	file name	pipe	file name	pipe

Poll Everywhere

pollev.com/cis5480

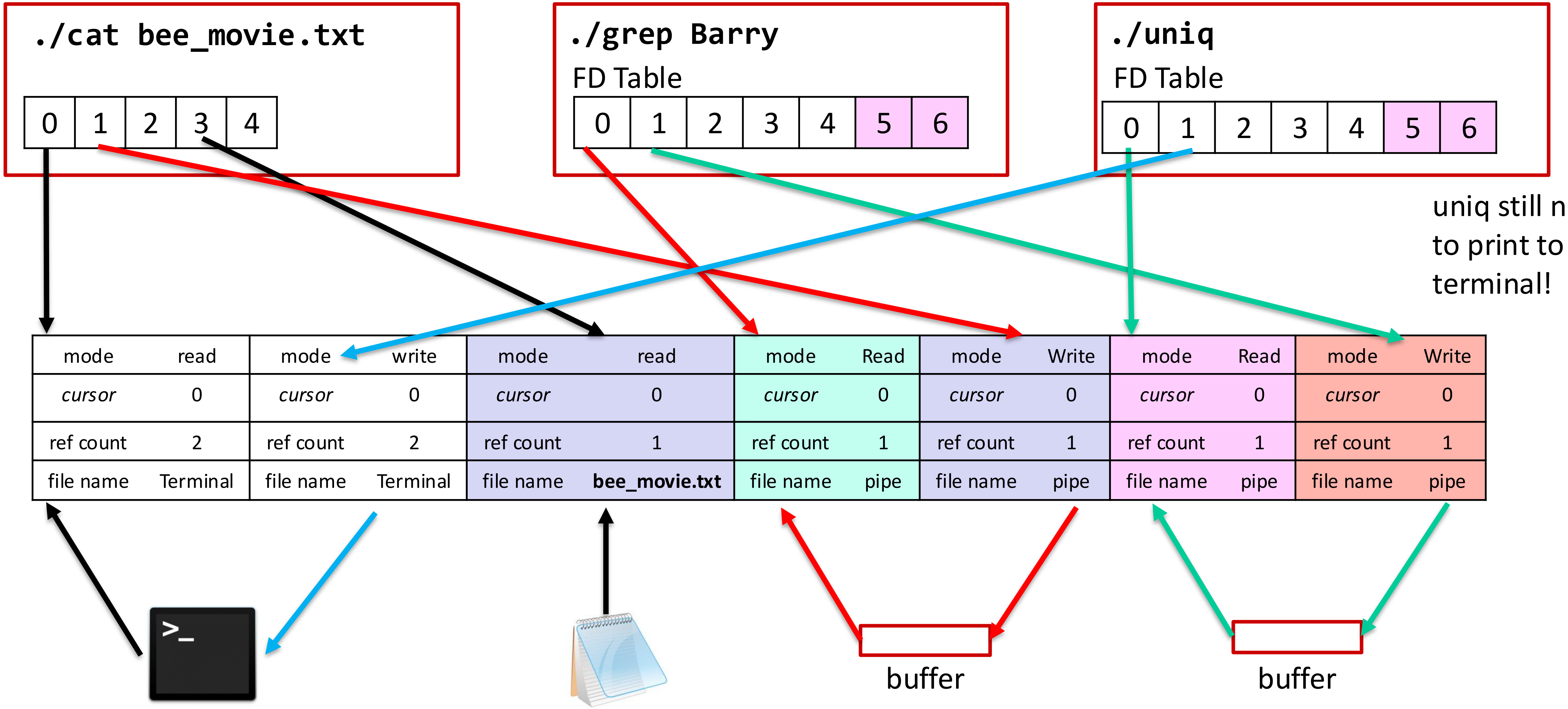
Why doesn’t uniq need to redirect it’s STDOUT?



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	**bee_movie.txt**	file name	pipe	file name	pipe	file name	pipe	file name	pipe

Poll Everywhere pollev.com/cis5480

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



```
./cat bee_movie.txt
```

0	1	2	3	4

```
./grep Barry
```

FD Table

0	1	2	3	4	5	6

```
./uniq
```

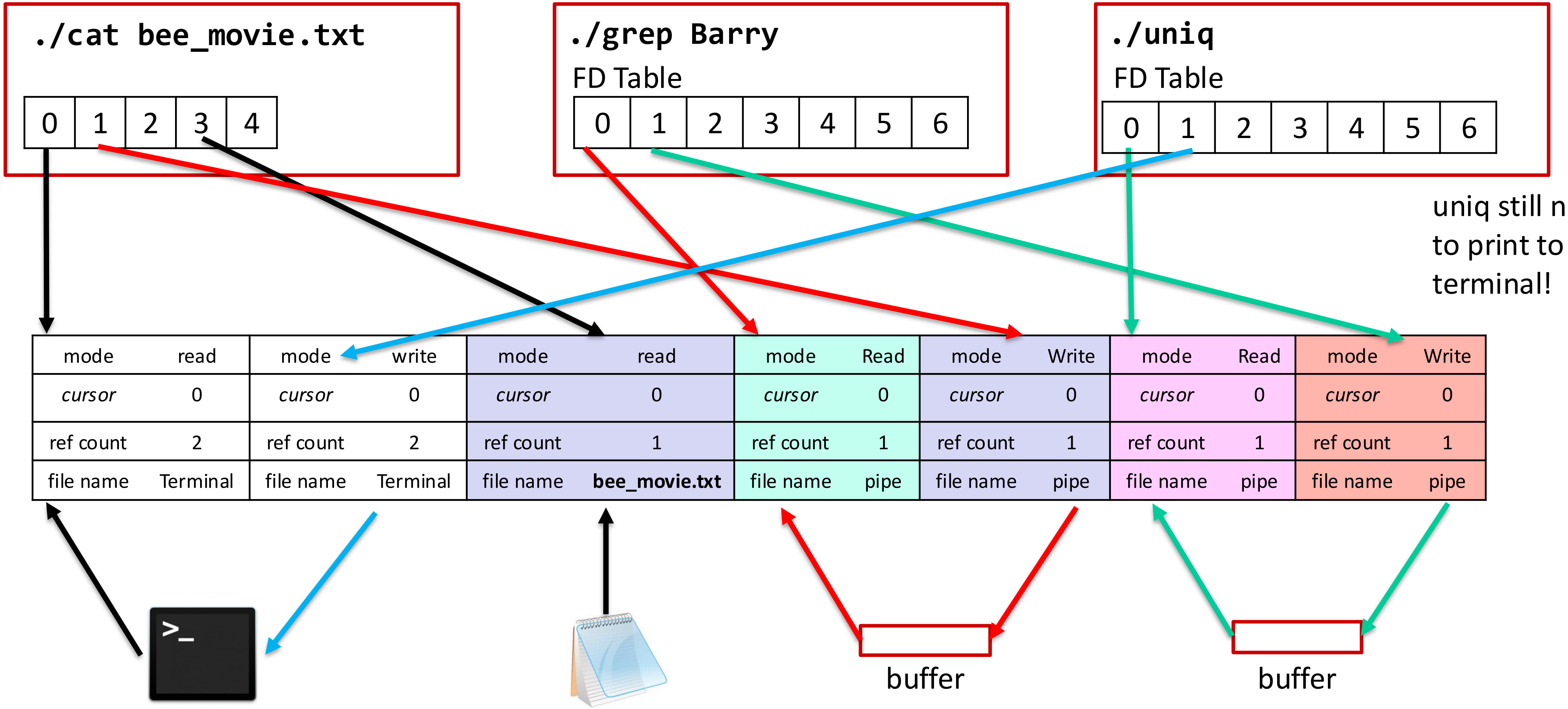
FD Table

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	**bee_movie.txt**	file name	pipe	file name	pipe	file name	pipe	file name	pipe

Let’s see it in code! Cool.



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	**bee_movie.txt**	file name	pipe	file name	pipe	file name	pipe	file name	pipe

uniq still needs to print to the terminal!

Poll Everywhere

pollev.com/cis5480

```
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?
}
pid_t uniq_pid = fork();
if(uniq_pid == 0){
   // do uniq stuff
 }
```

What could happen if you forget to close a write portion of the pipe, before EXEC-ing the grep?

`cat bee_movie.txt | grep Barry | uniq`

Forgetting to Close Pipes

```
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?
}
pid_t uniq_pid = fork();
if(uniq_pid == 0){
   // do uniq stuff
 }
```

`cat bee_movie.txt | grep Barry | uniq`

If you forget to close a file descriptor, ***especially those who share two pipes***, *then the program could very well stall. All due to one line mishap.*

Grep 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.

pipe2

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

- Still creates a pipe, similar to pipe, but we can now specify behavior!
- flags
  - O_CLOEXEC, your new friend.
  - This ***closes all file descriptors that refer to this pipe when we exec in a process.***
  - These file descriptors are only closed in the process that execs.
  - File descriptors that are ***dup2'd*** with these 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.

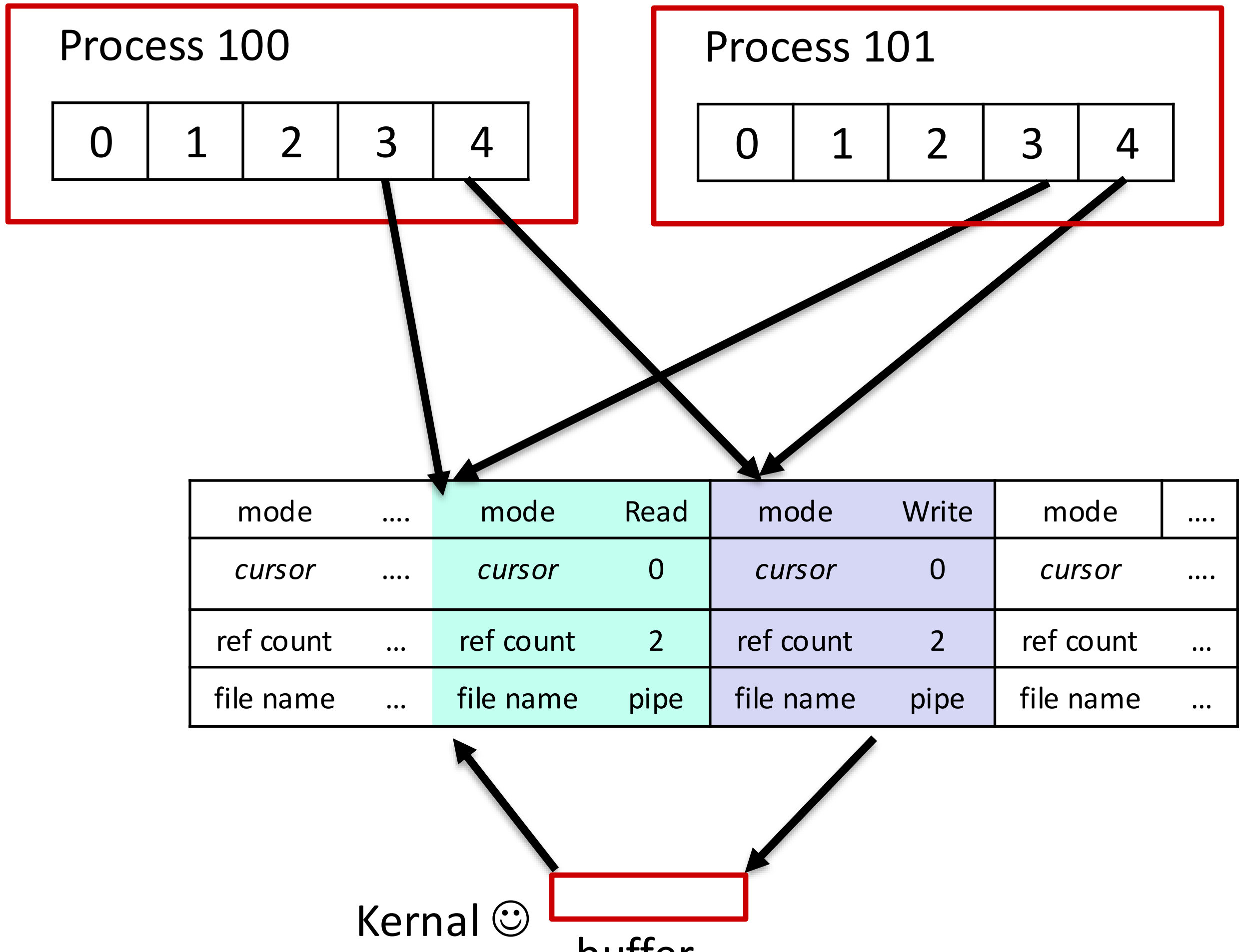
```
#define pipe2(FD, FLAG) \
pipe((FD)); \
fcntl((FD)[0], F_SETFD, FD_CLOEXEC); \
fcntl((FD)[1], F_SETFD, FD_CLOEXEC)
```

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!



Process 100

0	1	2	3	4

Process 101

0	1	2	3	4

mode	....	mode	Read	mode	Write	mode	....
cursor	....	*cursor*	0	*cursor*	0	*cursor*	....
ref count	...	ref count	2	ref count	2	ref count	...
file name	...	file name	pipe	file name	pipe	file name	...

Kernal ☺

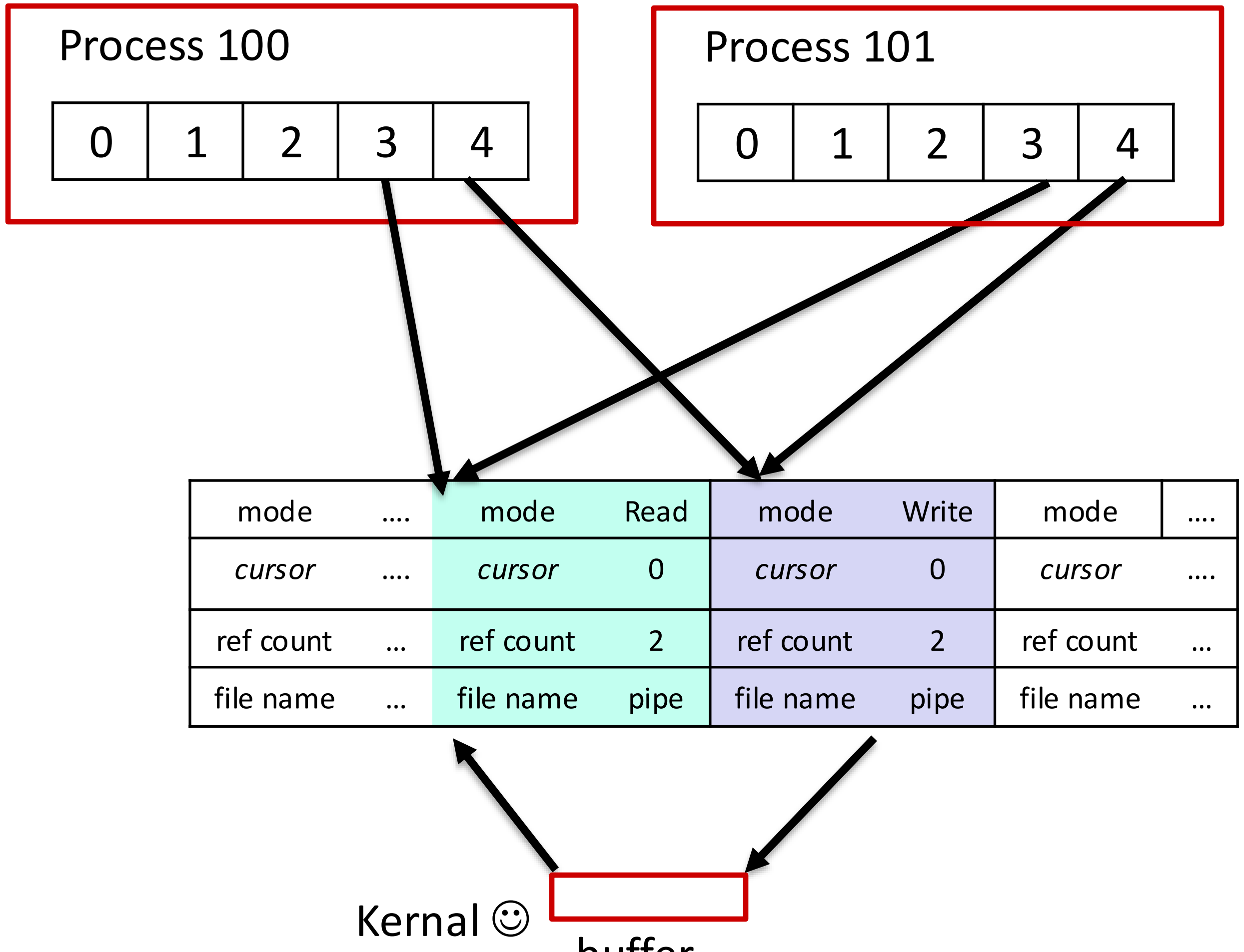
buffer

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!



Process 100

0	1	2	3	4

Process 101

0	1	2	3	4

mode	....	mode	Read	mode	Write	mode	....
cursor	....	*cursor*	0	*cursor*	0	*cursor*	....
ref count	...	ref count	2	ref count	2	ref count	...
file name	...	file name	pipe	file name	pipe	file name	...

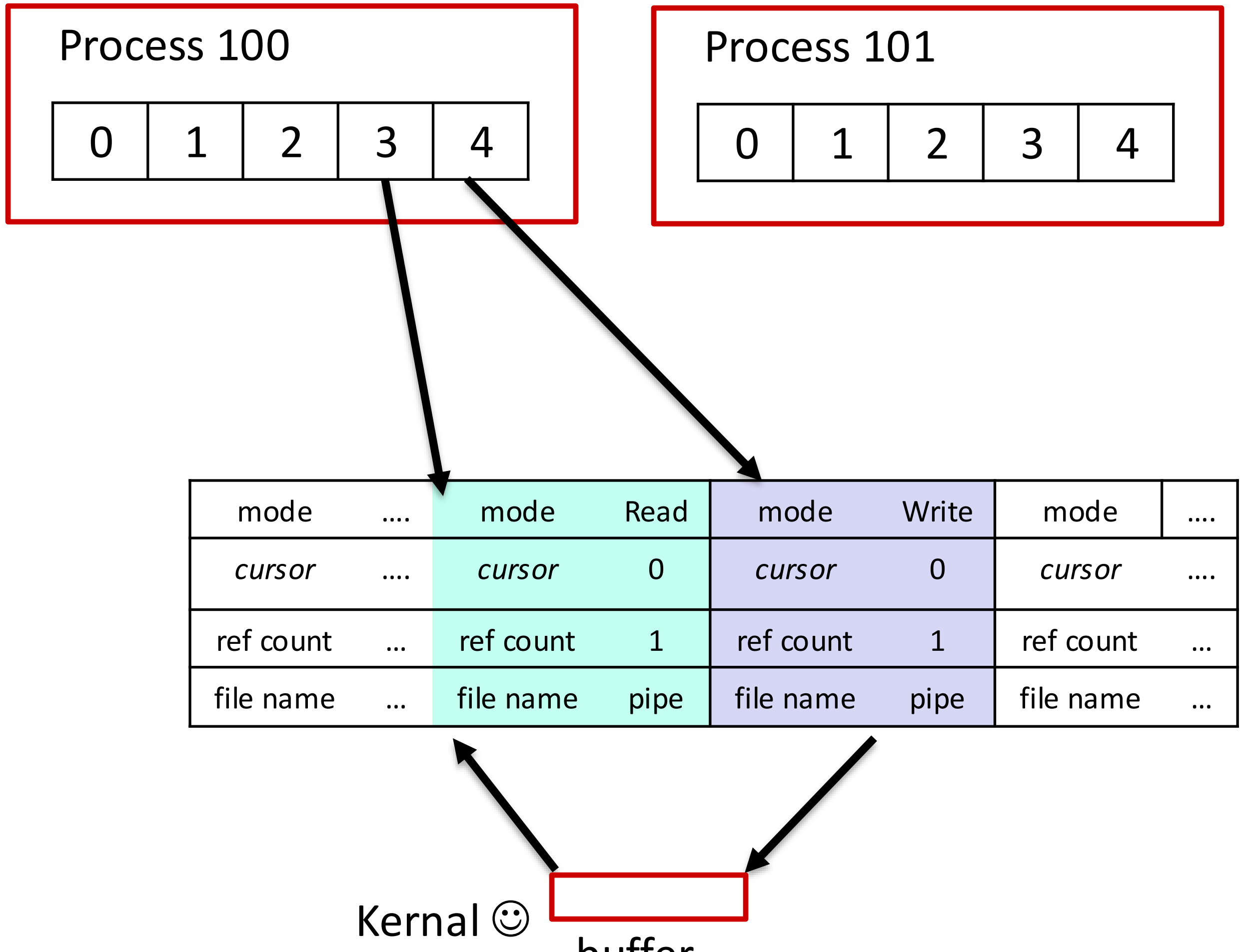
Kernal ☺

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!



Process 100

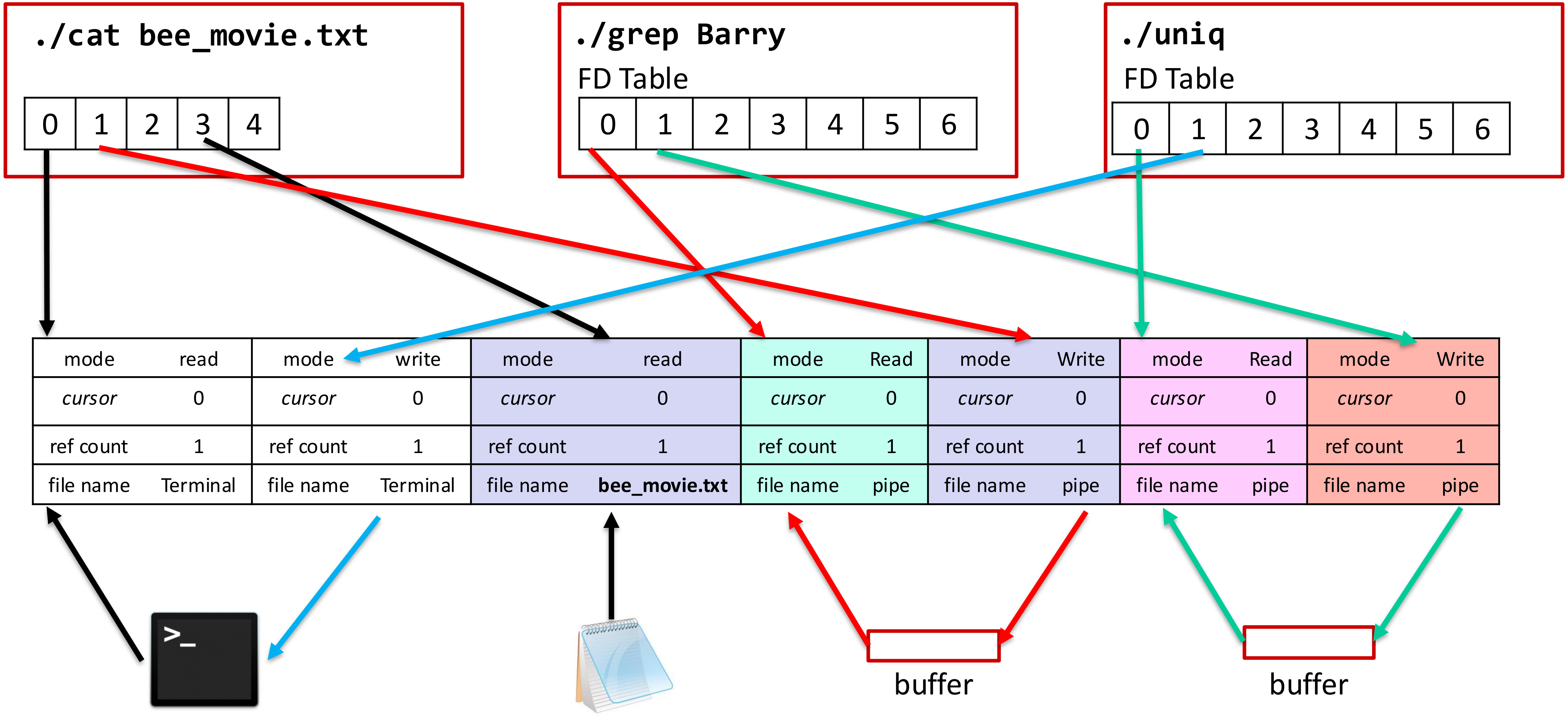
0	1	2	3	4

Process 101

0	1	2	3	4

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	...

Let’s see how pipe2 changes our code…



```
./cat bee_movie.txt
```

0	1	2	3	4

```
./grep Barry
```

FD Table

0	1	2	3	4	5	6

```
./uniq
```

FD Table

0	1	2	3	4	5	6

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	1	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	**bee_movie.txt**	file name	pipe	file name	pipe	file name	pipe	file name	pipe

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-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`"

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 bee_movie.txt > copy_bee_movie.txt
```

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.***

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

Poll Everywhere

pollev.com/cis5480

```
cat bee_movie.txt > copy_bee_movie.txt
```

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

```
char *bee_file_output = "copy_bee_movie.txt";

int bee_cpy_fd = open(bee_file_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).***"

Time for Penn Shell Demo!

- Ask Ash all questions. Don’t be shy pls.