OS: File Descriptors & Pipe()

Computer Systems Programming, Spring 2025

Instructor: Travis McGaha

Teaching Assistants:

Andrew Lukashchuk
Angie Cao
Aniket Ghorpade
Ashwin Alaparthi
Austin Lin
Hassan Rizwan
Lobi Zhao
Pearl Liu
Perrie Quek

pollev.com/tqm

- How are you?

Administrivia

- retry_shell (HW04)
  - Due 2/21 (Leaving open till 2/23)
  - Should have everything you need
  - Autograder and tests cases are out now
  - Leaving autograder open longer due to delay in getting it out

- pipe_shell (HW05)
  - To be released this week
  - Demo'd in recitation tomorrow
  - Should have everything you need after this lecture.
    Will have some more practice next week that may be helpful
  - Like retry shell, but instead of supporting a retry functionality, need to support piping between commands.

Lecture Outline

- File Descriptor Table & Redirections
- Pipe (start)
- Pipe motivation and in the shell
- Pipe Examples

open()/close()

- To open a file:
  - Pass in the filename and access mode
  - Get back a "file descriptor"
    - Similar to `FILE*` from `fopen()`, but is just an `int`
      - Returns `-1` to indicate error
    - Must manually close file when done ☹

Used to identify a file w/ the OS

```
#include <fcntl.h>     // for open()
#include <unistd.h>    // for close()
  ...
  int fd = open("foo.txt", O_RDONLY);
  if (fd == -1) {
    perror("open failed");
    exit(EXIT_FAILURE);
  }
  ...
  close(fd);
```

Reading from a File

Stores read result in buf

Number of bytes

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

signed

- Function is written in C: follows C design
  - Takes in a file descriptor
  - Takes in an array and length of where to store the results of the read
  - Returns number of bytes read

- EVERY TIME we read from a file, this function is getting called somewhere
  - Even in Java or Python
  - There are wrappers around this, but they are all implemented on top of these system calls
  - The OS doesn't speak java or python, it "speaks" assembly and C so all languages must have a way to invoke these C functions.

Going over this quickly: the important point is not to memorize this function; we will go over it again later.

The main thing is this: whenever we interact with a file (even in other languages) somewhere under the hood it is calling these C functions

stdout, stdin, stderr

- By default, there are three “files” open when a program starts
  - stdin: for reading terminal input typed by a user
    - `cin` in C++
    - `System.in` in Java
  - stdout: the normal terminal output.
    - `cout` in C++
    - `System.out` in Java
  - stderr: the terminal output for printing errors
    - `cerr` in C++
    - `System.err` in Java

stdout, stdin, stderr

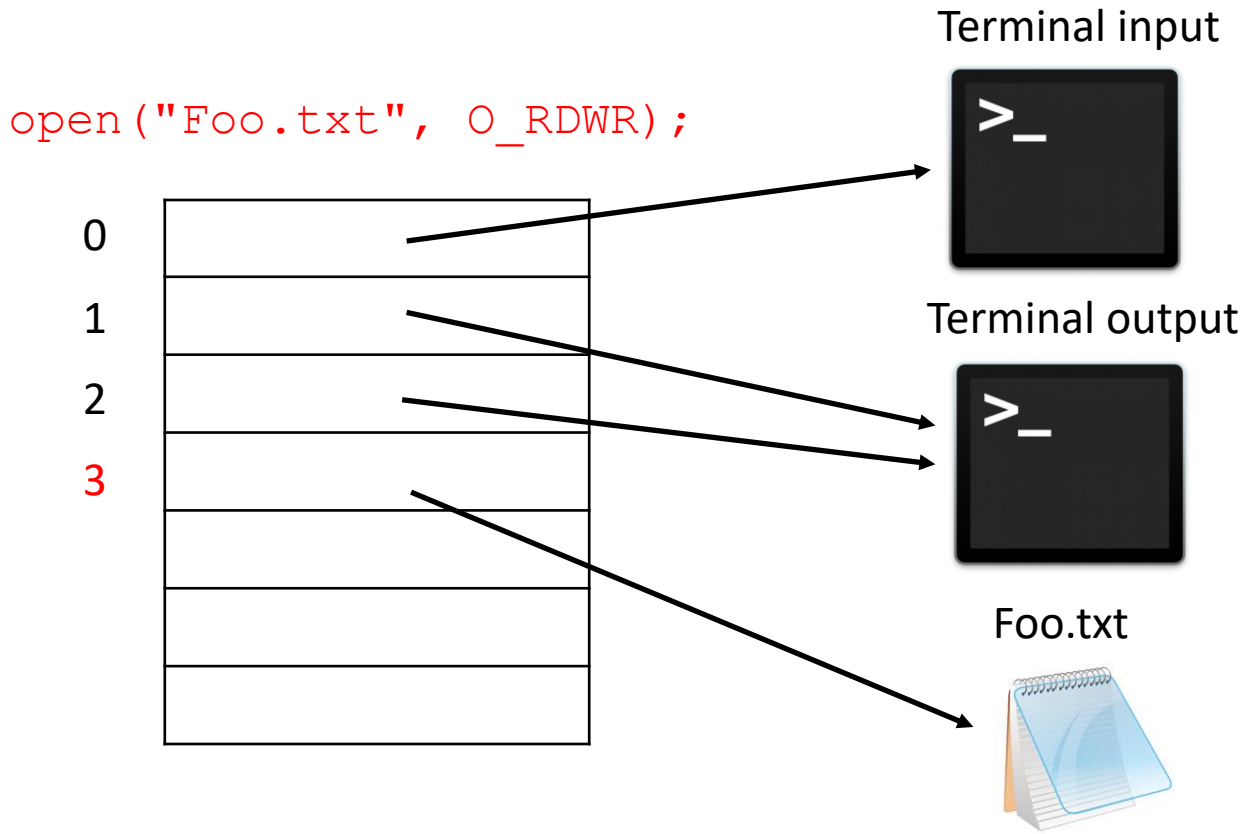
- stdin, stdout, and stderr all have initial file descriptors constants defined in `unistd.h`
  - `STDIN_FILENO  -> 0`
  - `STDOUT_FILENO  -> 1`
  - `STDERR_FILENO  -> 2`

- These will be open on default for a process

- Printing to stdout with cout will use `write(STDOUT_FILENO, …)`

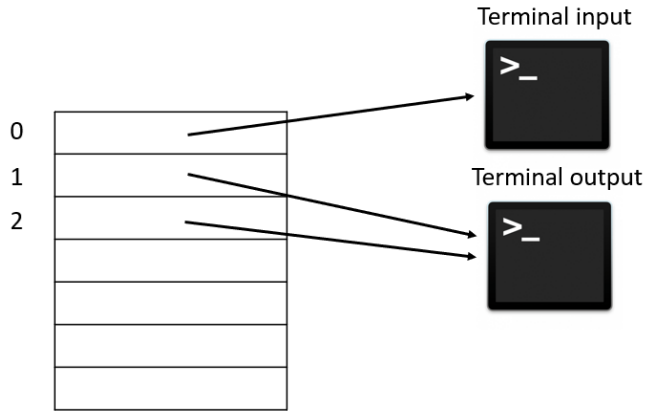
File Descriptor Table

- In addition to an address space, each process will have **its own file descriptor table** managed by the OS
- The table is just an array, and the file descriptor is an index into it.



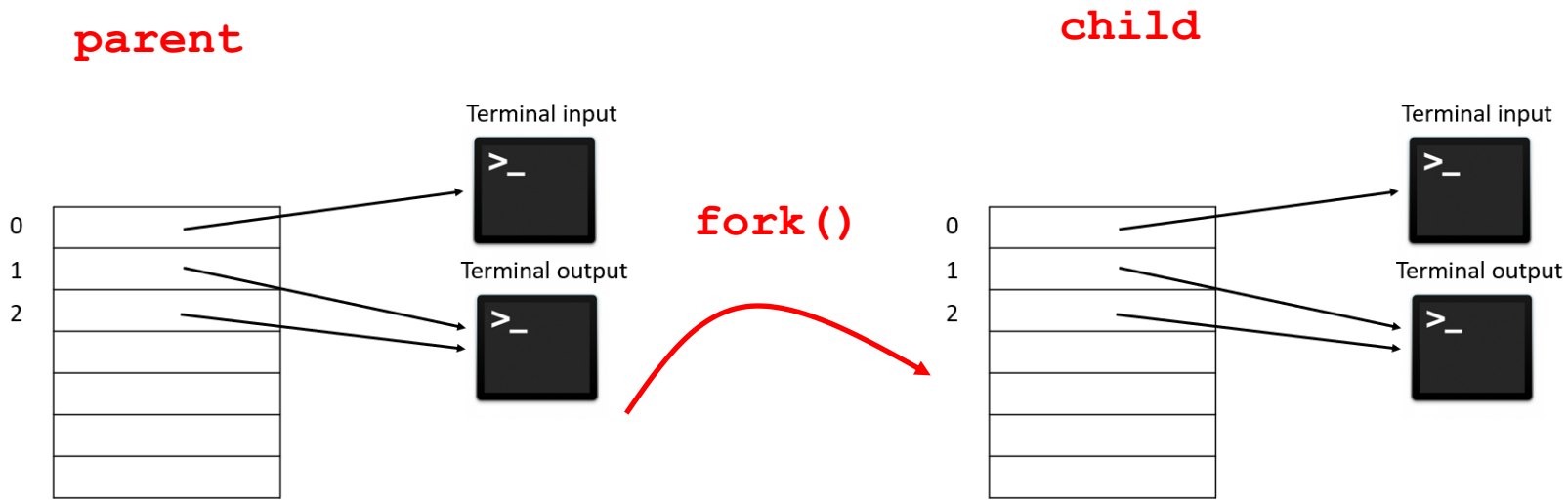
File Descriptor Table: Per Process

- 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



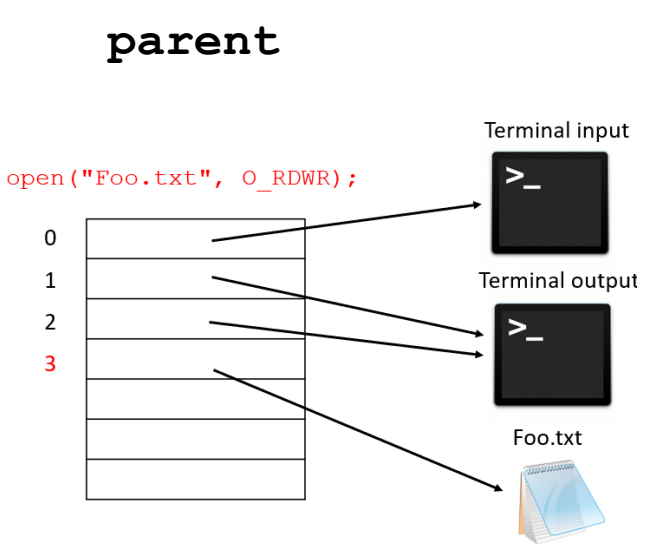
File Descriptor Table: Per Process

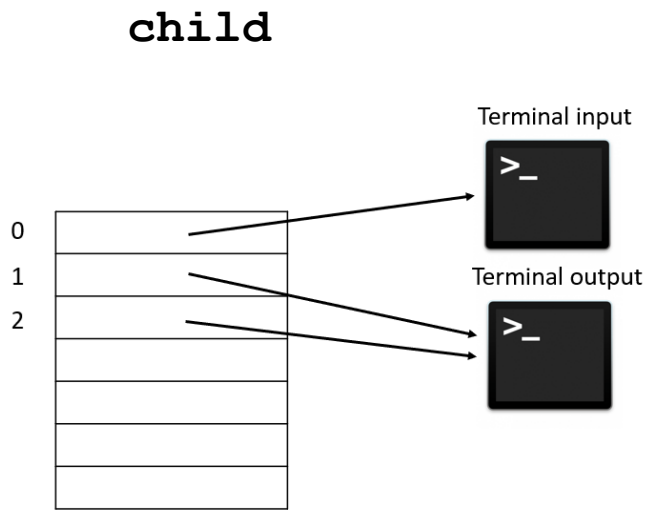
- each process will have **its own file descriptor table** managed by the OS
- Fork will make a copy of the parent's file descriptor table for the child



File Descriptor Table: Per Process

- 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





Child is unaffected by parent calling open!

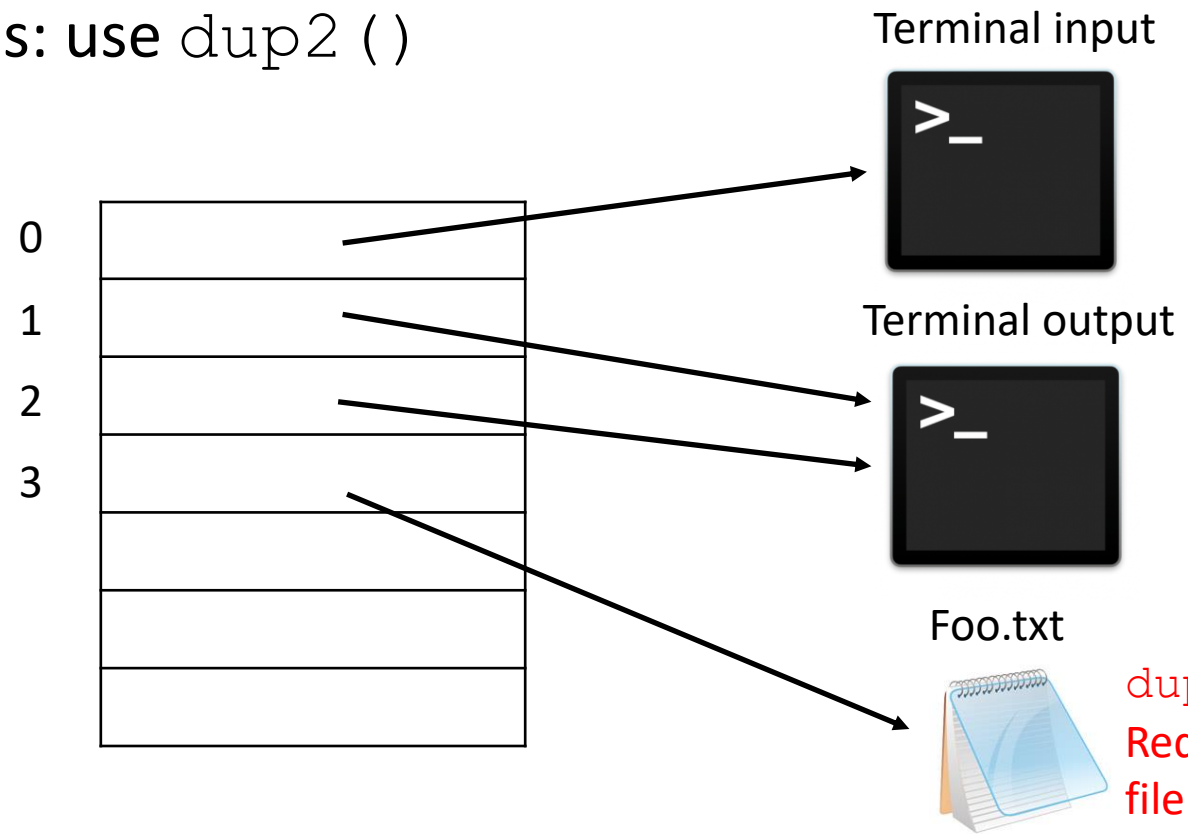
Gap Slide

- Gap slide to distinguish we are moving on to a new example (that looks very similar to the previous one)

Redirecting stdin/out/err

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

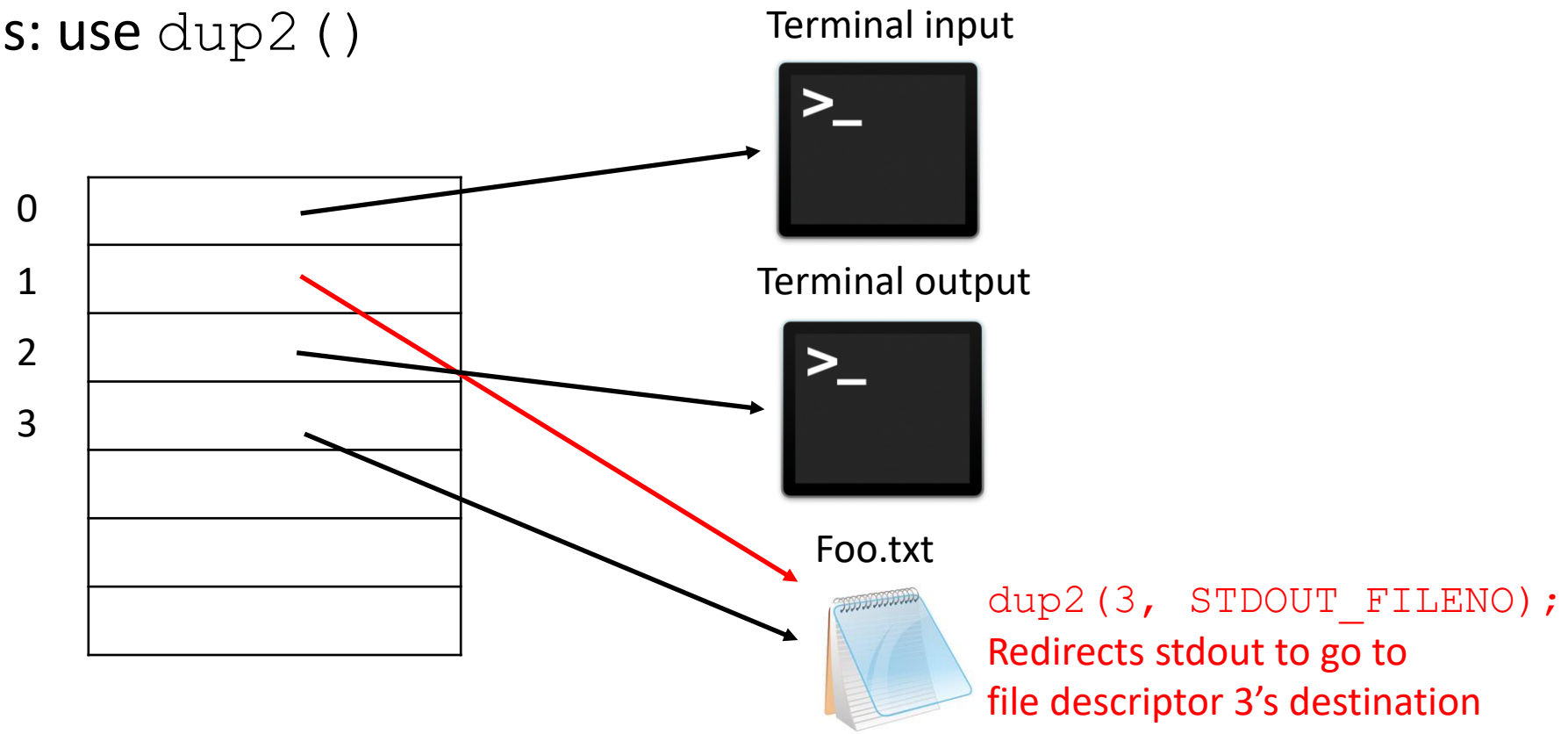
- We can change things so that STDOUT_FILENO is associated with something other than a terminal output.
- Now, any calls to printf, cout, System.out, etc now go to the redirected output
- To do this: use `dup2()`



`dup2(3, STDOUT_FILENO);`
Redirects stdout to go to file descriptor 3's destination

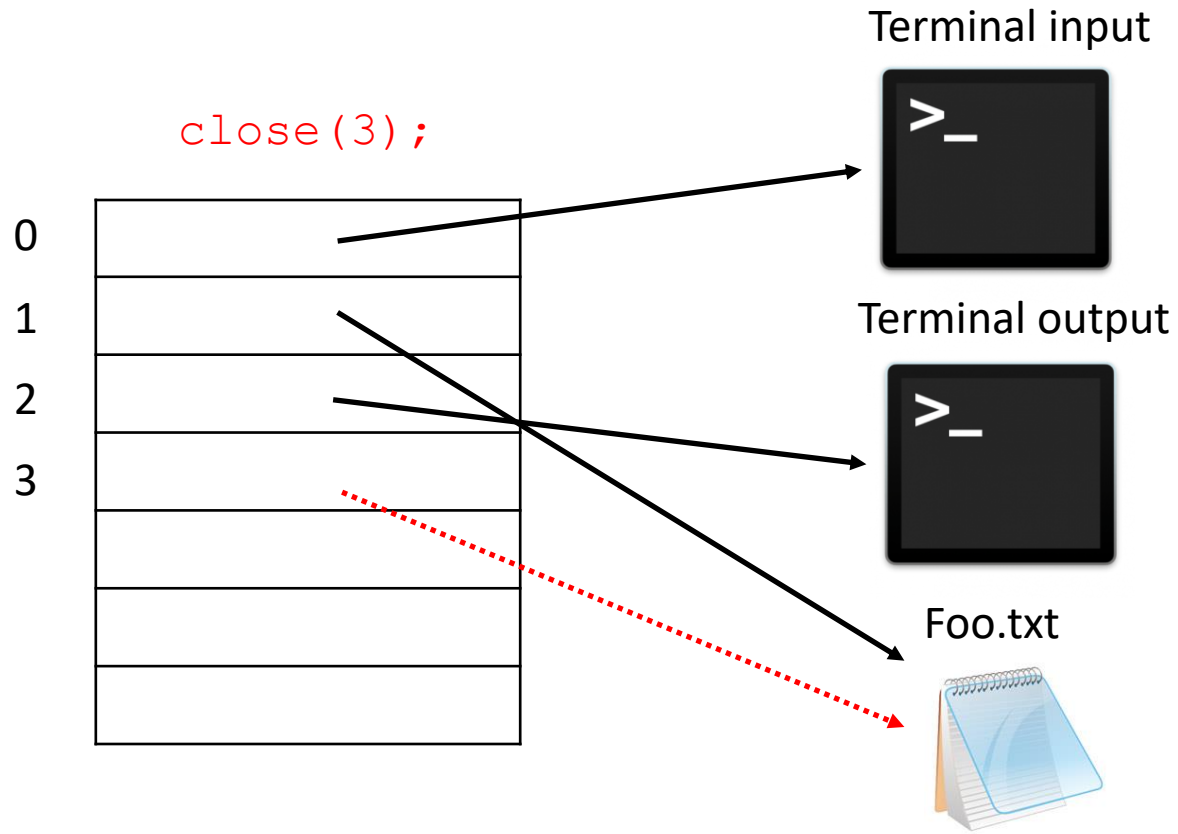
Redirecting stdin/out/err

- We can change things so that STDOUT_FILENO is associated with something other than a terminal output.
- Now, any calls to printf, cout, System.out, etc now go to the redirected output
- To do this: use `dup2()`



Closing a file descriptor

- If we close a file descriptor, it only closes that descriptor, not the file itself
- Other file descriptors to the same file will still be open
- use `close()`



dup2()

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

  File descriptor

  - Creates a copy of the file descriptor **`oldfd`** using **`newfd`** as the new file descriptor number
  - If **`newfd`** was a previously open file, it is silently closed before being reused
  - Returns -1 on error.

Poll Everywhere

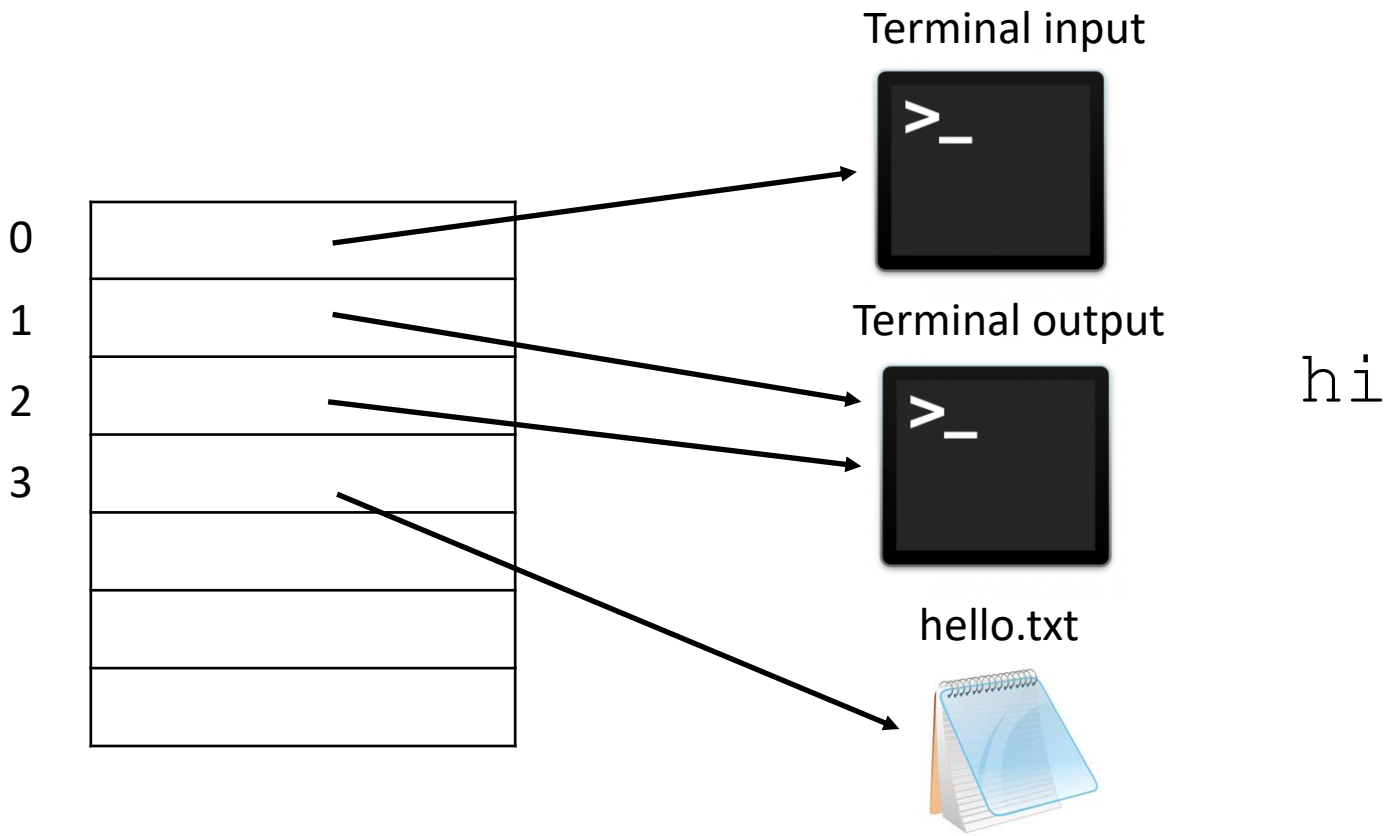
pollev.com/tqm

- Given the following code, what is the contents of "hello.txt" and what is printed to the terminal?

```
 9 int main() {
10   int fd = open("hello.txt", O_WRONLY);
11
12   printf("hi\n");
13
14   close(STDOUT_FILENO);
15
16   printf("?\n");
17
18   // open `fd` on `stdout`
19   dup2(fd, STDOUT_FILENO);
20
21   printf("!\n");
22
23   close(fd);
24
25   printf("*\n");
26
27 }
```

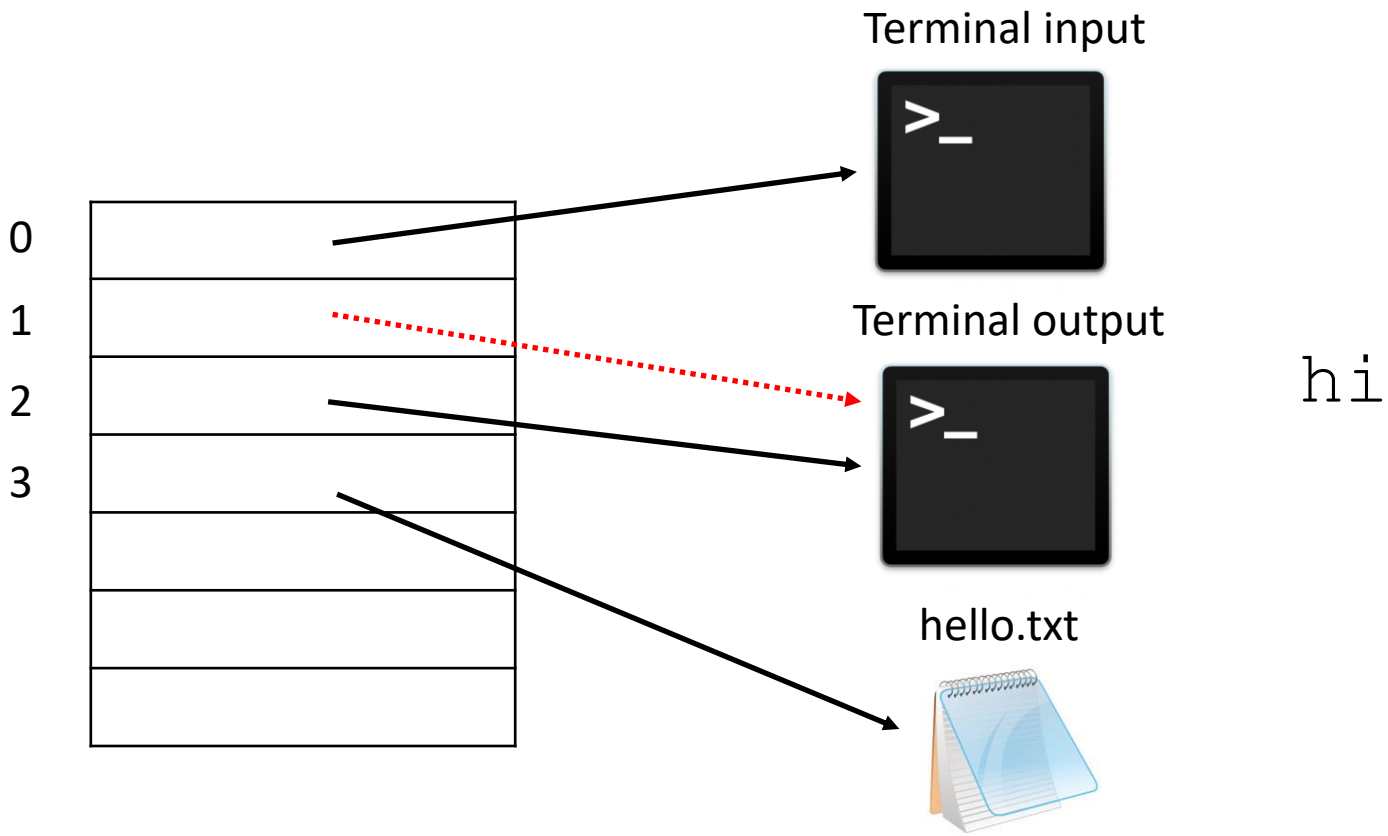
Explanation

```
int fd = open("hello.txt", O_WRONLY);
printf("hi\n");
```



Explanation

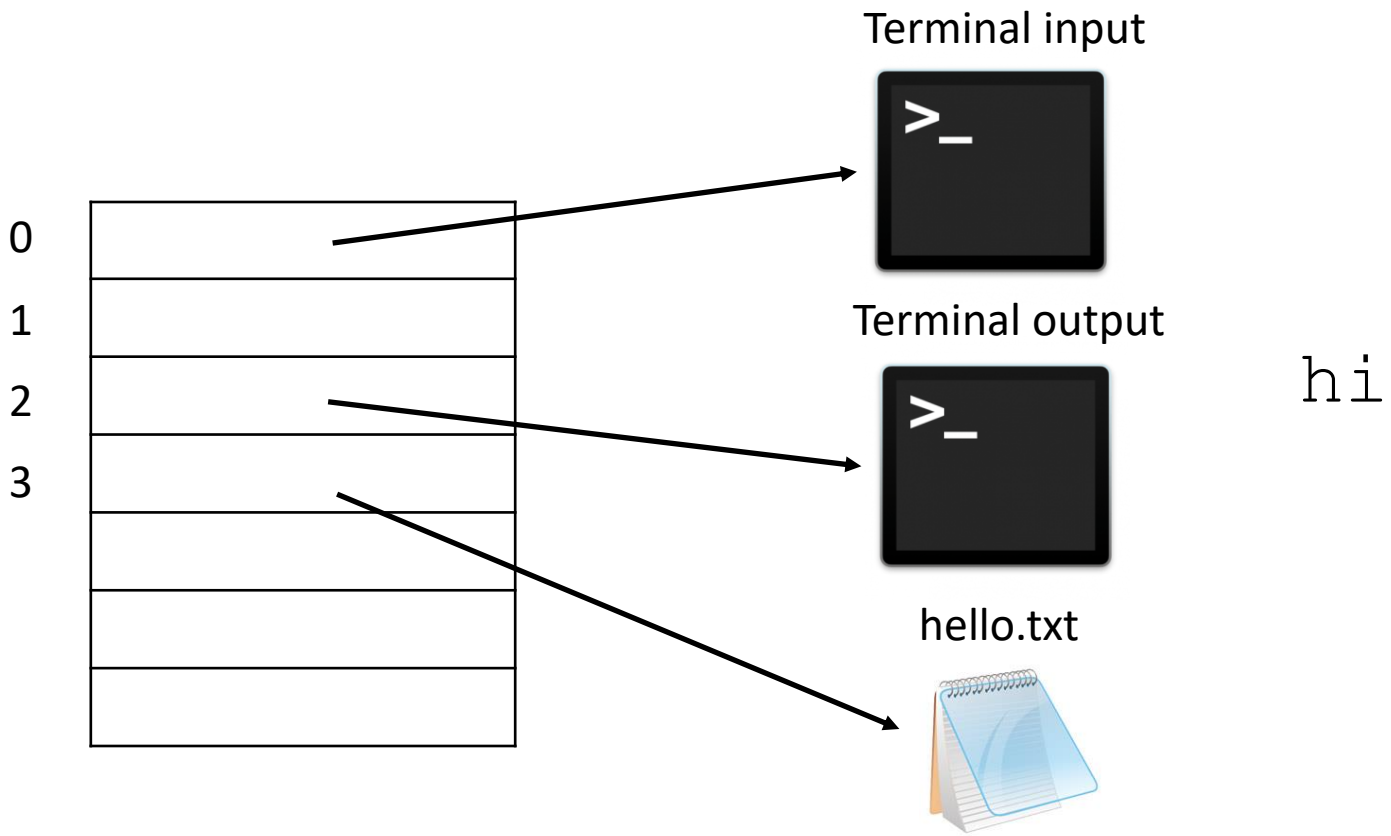
```
close(STDOUT_FILENO);
```




```
hi
```

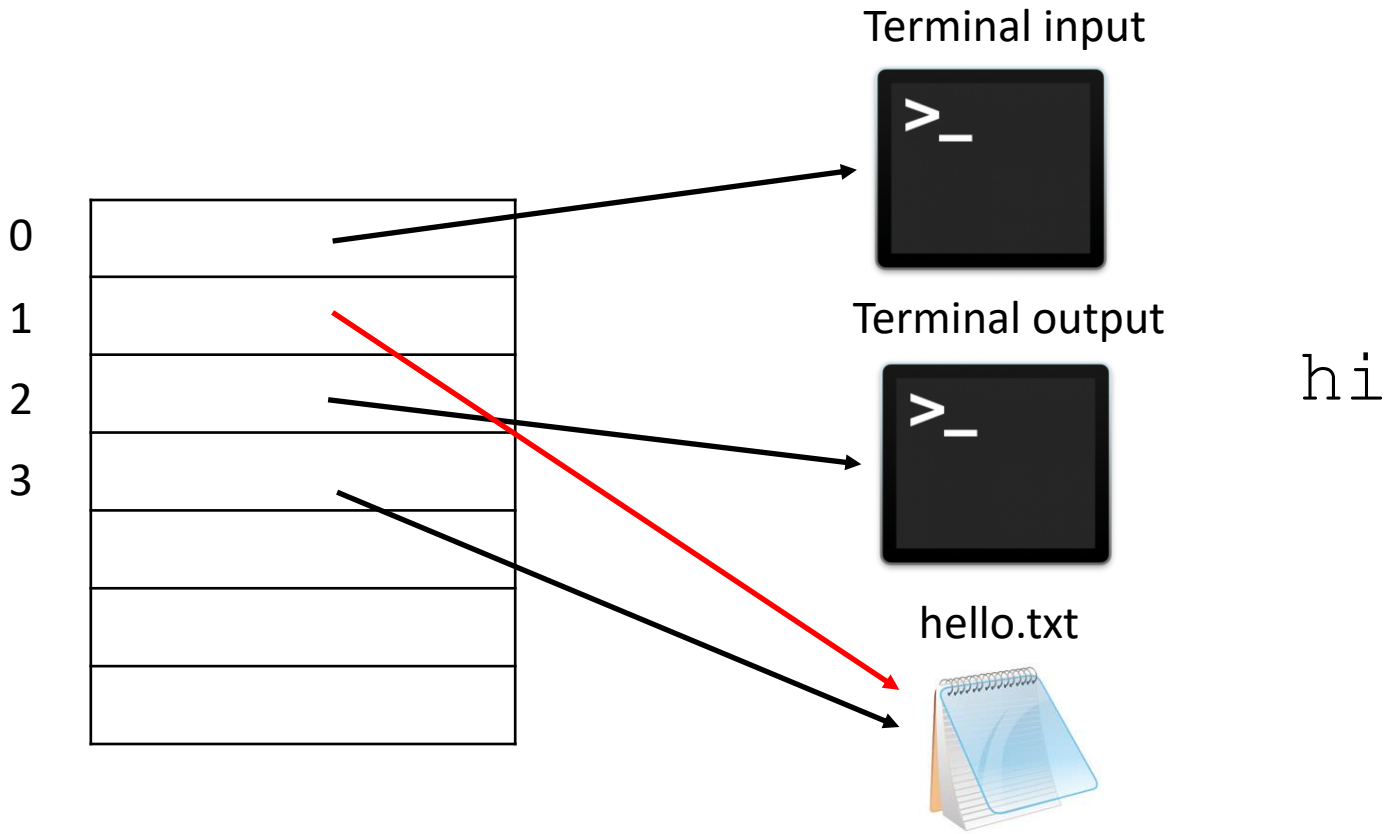
Explanation

```
close(STDOUT_FILENO);
printf("?\n");          // errors! Nothing printed
```



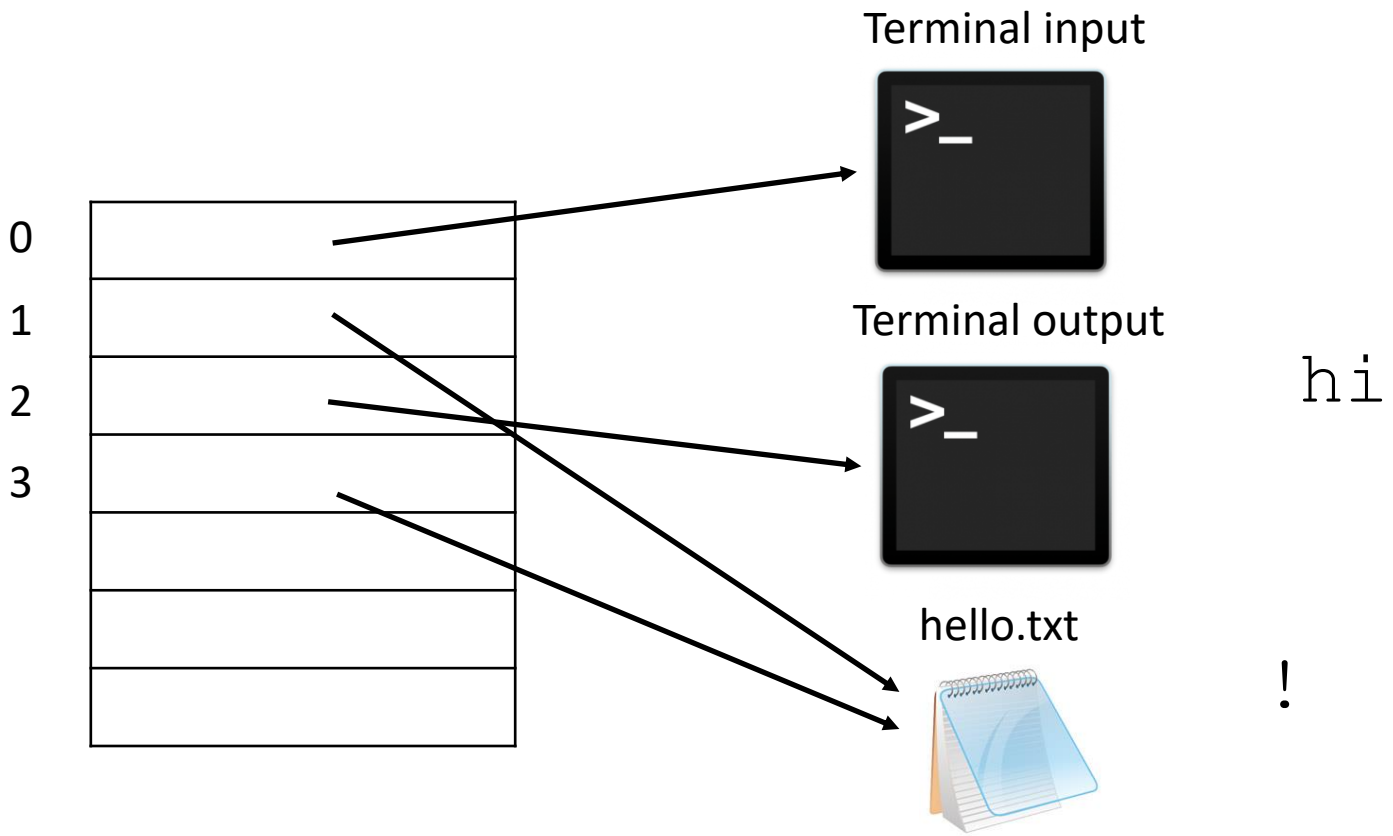
Explanation

```
dup2(fd, STDOUT_FILENO);
```



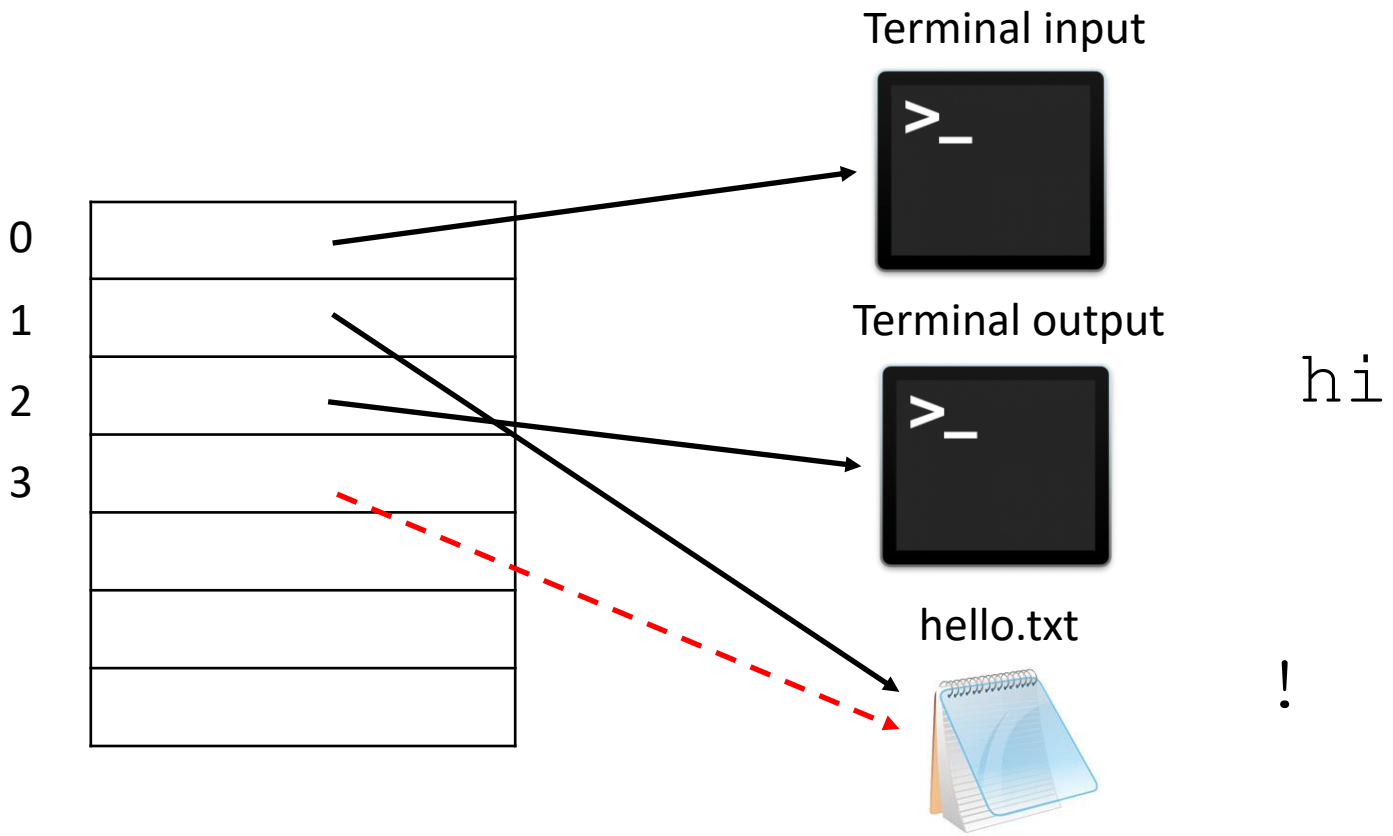
Explanation

```
dup2(fd, STDOUT_FILENO);
printf("!\n");
```



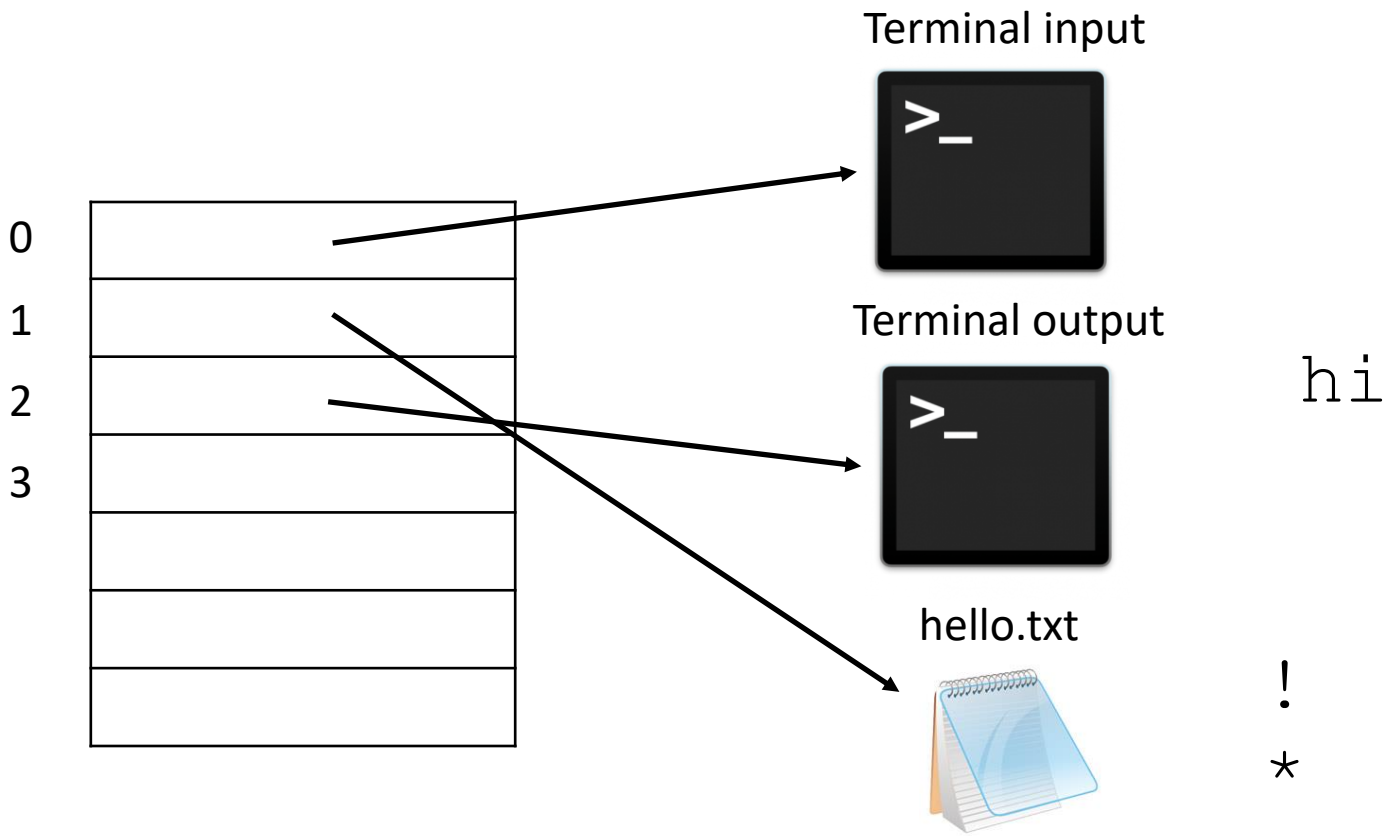
Explanation

```
close(fd);
```



Explanation

```
printf("*\n");
```



Lecture Outline

Pipes

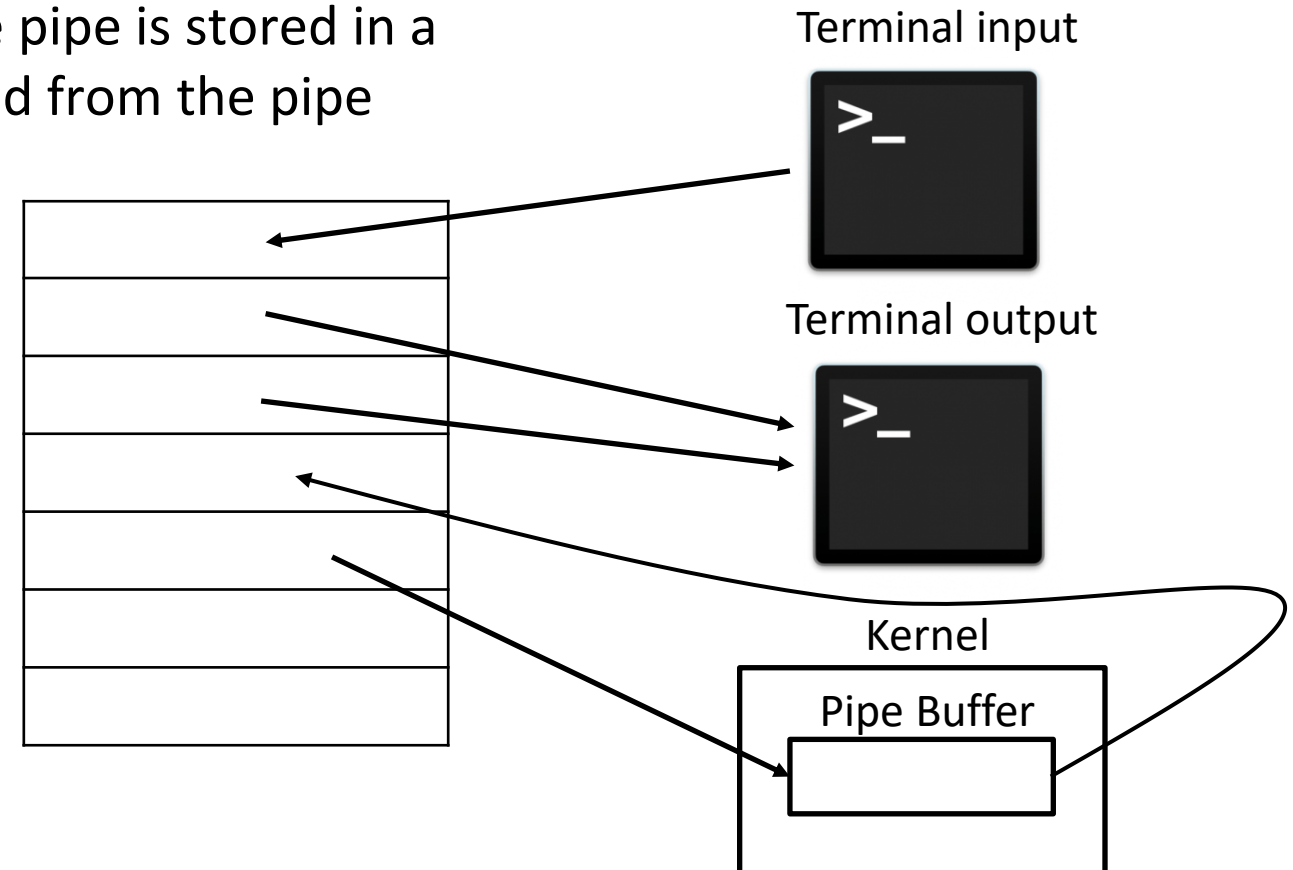
```
int pipe(int pipefd[2]);
```

- 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

- **In addition to copying memory, fork copies the file descriptor table of parent**
- Exec does NOT reset file descriptor table

Pipe Visualization

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



I/O "Streams"

- The way files are stored is quite complicated (see CIS 5480).
  But from a user level program, we have a nice "stream" abstraction.

- A stream is a linear sequence of bytes/characters that we can read bytes from or write bytes too.
  - We don't have to worry about the time it takes to read the file (unless we want to)
  - We don't have to worry about how bytes of a file may not be stored "in order" in the filesystem
  - We don't know the "length" of the stream until it ends and we hit EOF

- Is a metaphor similar to how there is a "Stream" of water. The water flows nicely from one point to another.

EOF & Streams

- How reading and writing to streams can vary a lot based on what our "stream" is over. These details are mostly hidden from you.
  - Is this a stream for just reading a file?
  - Is this a stream for reading data over the network?
  - Is this a stream for reading from a pipe?
  - Something else?
  - In Linux and UNIX-like systems there can be some small differences, but they all act mostly like reading or writing a file.

- What is EOF? **E**nd-**O**f-**F**ile. Indicates that there is nothing left to read from a stream. When do we hit EOF when reading a file?

Pipes & EOF

- Many programs will read from a file until they hit EOF and will not terminate until then
- Like reading from the terminal, just because there is nothing in the pipe, does not mean nothing else will ever come through the pipe.
  - EOF is not read in this case

- EOF is only read from a pipe when:
  - There is nothing in the pipe
  - All write ends of the pipe are closed

- **Good practice: CLOSE ALL PIPE FDS YOU ARE DONE WITH**

Poll Everywhere

pollev.com/tqm

- What does the parent print? What does the child print? why? (assume pipe, close and fork succeed). Note: code has some bad practices

```
12 // writes the string to the specified fd
13 bool wrapped_write(int fd, const string& to_write);
14
15 // reads till eof from specified fd. nullopt on error
16 optional<string> wrapped_read(int fd);
17
18 int main() {
19   array<int, 2> pipe_fds;
20   pipe(pipe_fds.data());
21
22   // child process only exits after this
23   pid_t pid = fork();
24
25   if (pid == 0) {
26     // child process
27
28     // close the end of the pipe that isn't used
29     close(pipe_fds.at(0));
30
31     string greeting {"Hello!"};
32     wrapped_write(pipe_fds.at(1), greeting);
33
34     optional<string> response = wrapped_read(pipe_fds.at(1));
35
36     if (response.has_value()) {
37       cout << response.value() << endl;
38     }
39
40     exit(EXIT_SUCCESS);
41   }
42   // parent
```

pipe_unidirect.cpp
on course website

```
42   // parent
43
44   /// close the end of the pipe I won't use
45   close(pipe_fds.at(1));
46
47   optional<string> message = wrapped_read(pipe_fds.at(0));
48
49   if (message.has_value()) {
50     cout << message.value() << endl;
51   }
52
53   string greeting{"Howdy!"};
54   wrapped_write(pipe_fds.at(0), greeting);
55
56   int wstatus;
57   waitpid(pid, &wstatus, 0);
58
59   return EXIT_SUCCESS;
60 }
```

Pipes & EOF

- Many programs will read from a file until they hit EOF and will not terminate until then
- Like reading from the terminal, just because there is nothing in the pipe, does not mean nothing else will ever come through the pipe.
  - EOF is not read in this case

- EOF is only read from a pipe when:
  - There is nothing in the pipe
  - All write ends of the pipe are closed

- **Good practice: CLOSE ALL PIPE FDS YOU ARE DONE WITH**

Lecture Outline

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`" and "`echo hello | cat | wc -l`" DEMO

- `cmd > file`, redirects the stdout of a command to be written to the specified file
- Complex example:

```
cat ./input.txt | ./retry_shell > 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.

Lecture Outline

Unix Shell Control Operators: Pipe

- `cmd1 | cmd2`, creates a pipe so that the stdout of cmd1 is redirected to the stdin of cmd2
  - E.g. "`cat ./test_files/mutual_aid.txt | grep communism`"

pipe_shell overview

- In pipe_shell, you will be writing your own shell that reads from user input
  - Each line is a command that could consist of multiple programs and pipes between them
  - Your shell should fork a process to run each program and setup the pipes in between them

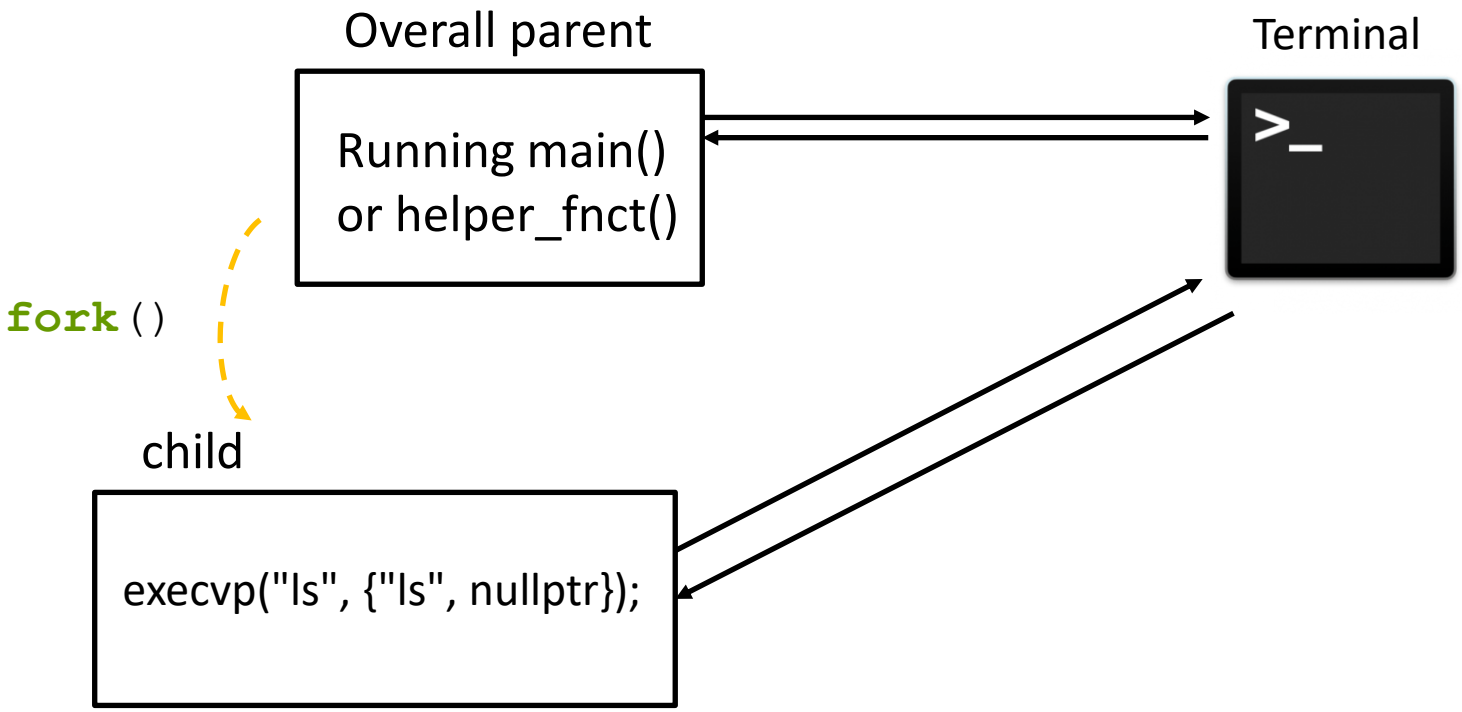
- Some sample programs provided to help with implementation ideas.

Suggested Approach

- HIGHLY ENCOURAGED to follow the suggested approach
  - Write a program that implements the basic functionality of retry_shell (no retrying needed)
  - Make sure that it can handle commands with no pipes
    - `"ls"`
  - Make sure that it can handle command line arguments
    - `"ls -l"`
  - Add support for commands with ONE pipe
    - `"ls -l | wc"`
  - Generalize to add support for any number of pipes
    - `"ls -l | wc | cat"`

pipe_shell Example Line

- Consider the case when a user inputs
  - `"ls"`



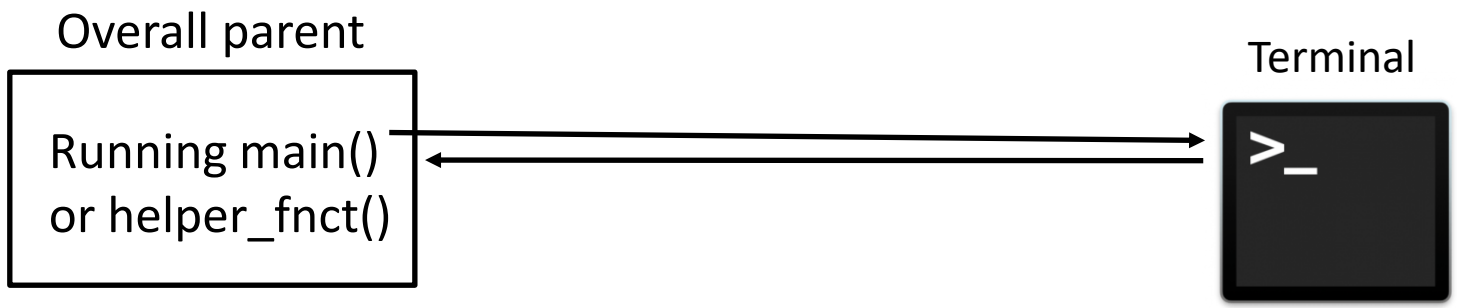
pipe_shell Hints

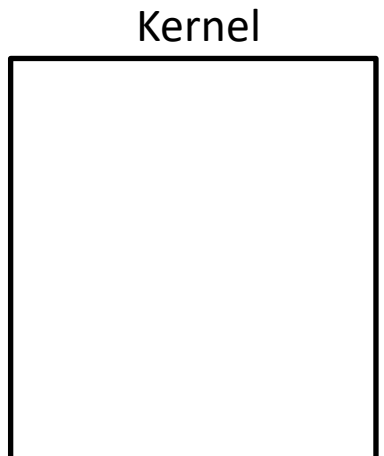
- If there are n commands in a line, there should be n-1 pipes
- Each pipe should be written to by exactly one process
- Each pipe should be read by exactly one process
  - Different than the one writing
- There are three cases to consider for commands using pipes
  - The first process, which reads from stdin and writes out to a pipe
  - The last process, which reads from a pipe and writes to stdout
  - Processes in between which read from one pipe and write to another

- More hints when HW is posted

pipe_shell Example Line 1

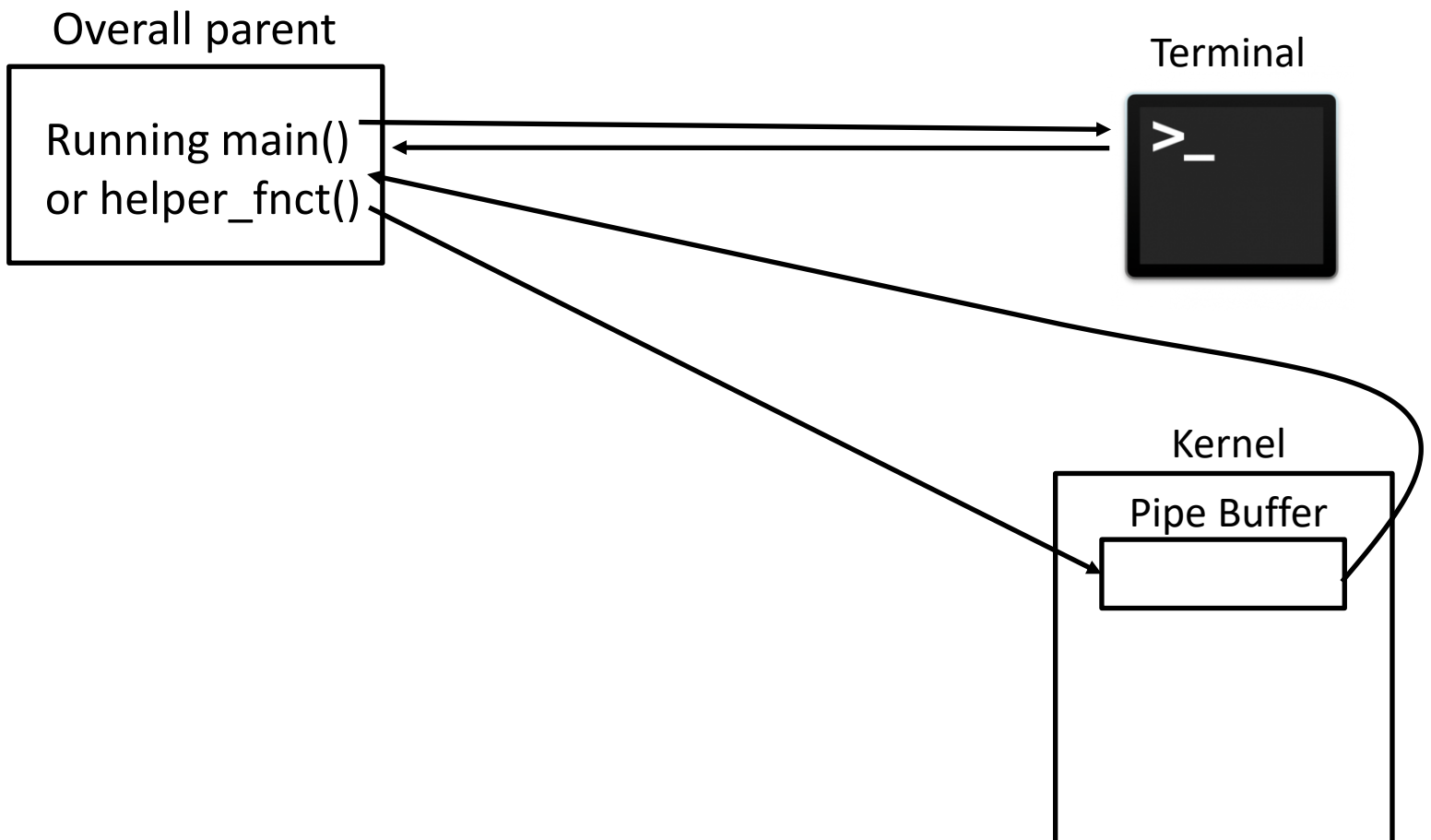
- Consider the case when a user inputs
  - `"ls | wc"`





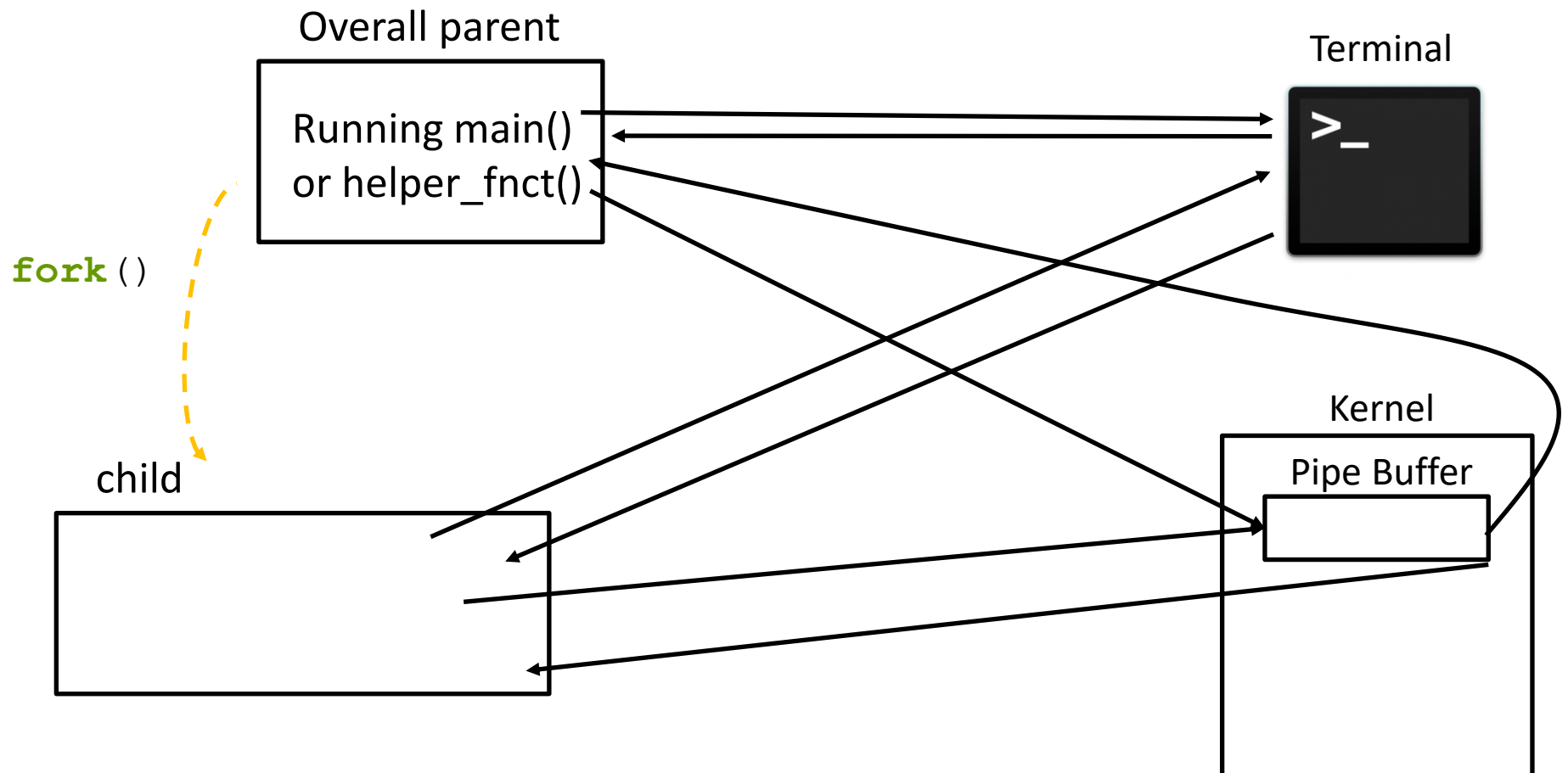
pipe_shell Example Line 1

- Consider the case when a user inputs
  - `"ls | wc"`



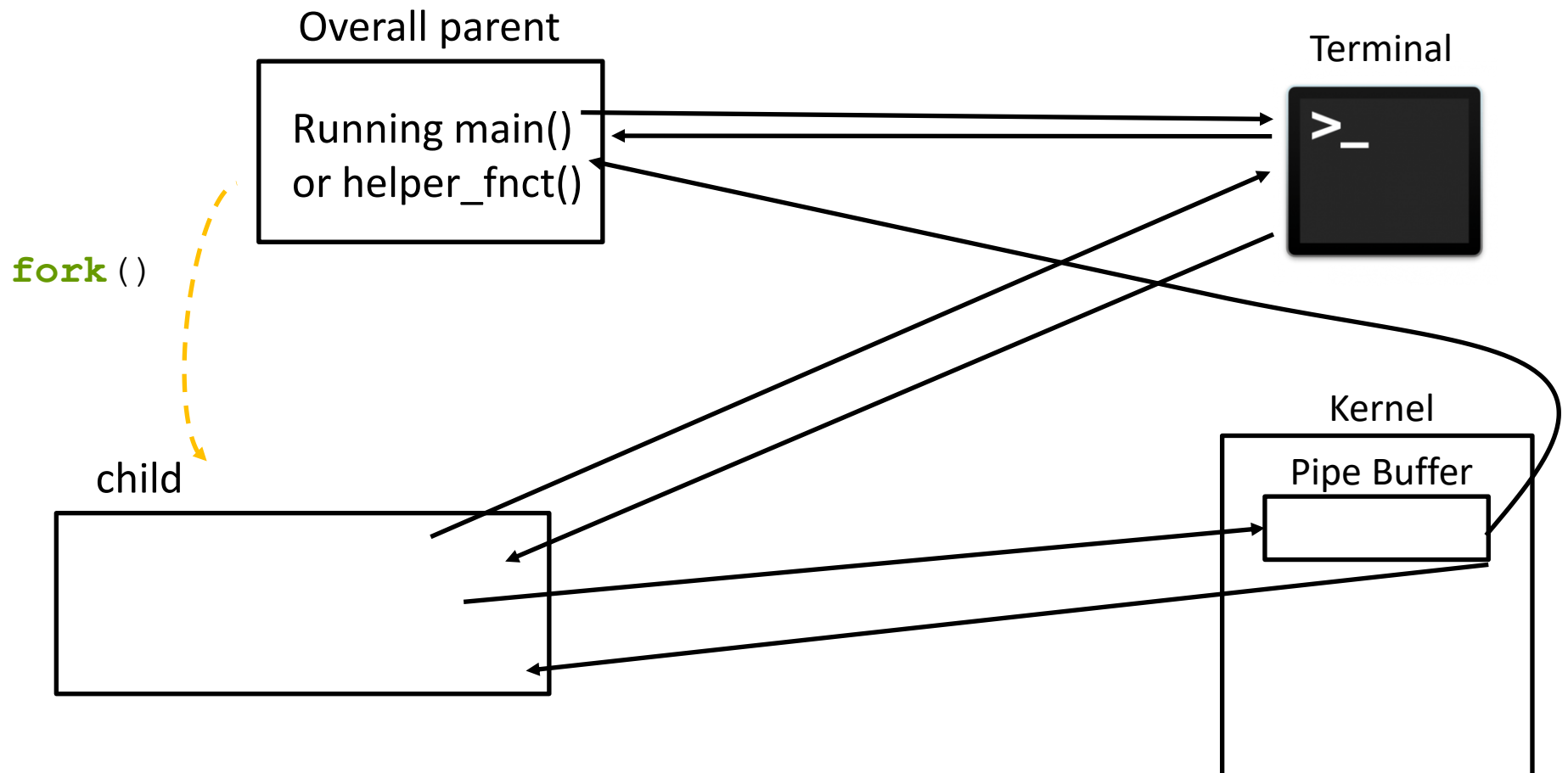
pipe_shell Example Line 1

- Consider the case when a user inputs
  - `"ls | wc"`



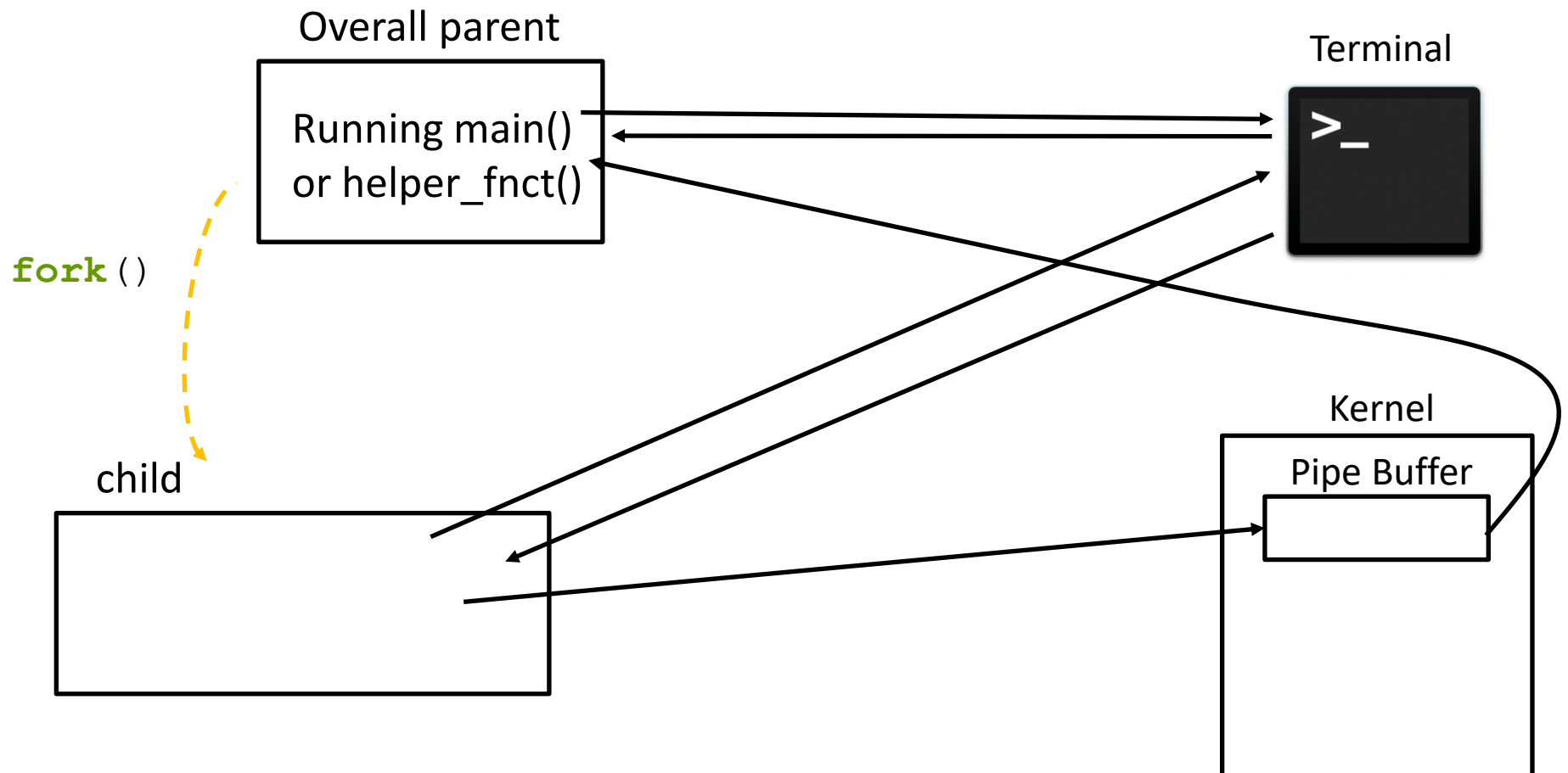
pipe_shell Example Line 1

- Consider the case when a user inputs
  - `"ls | wc"`



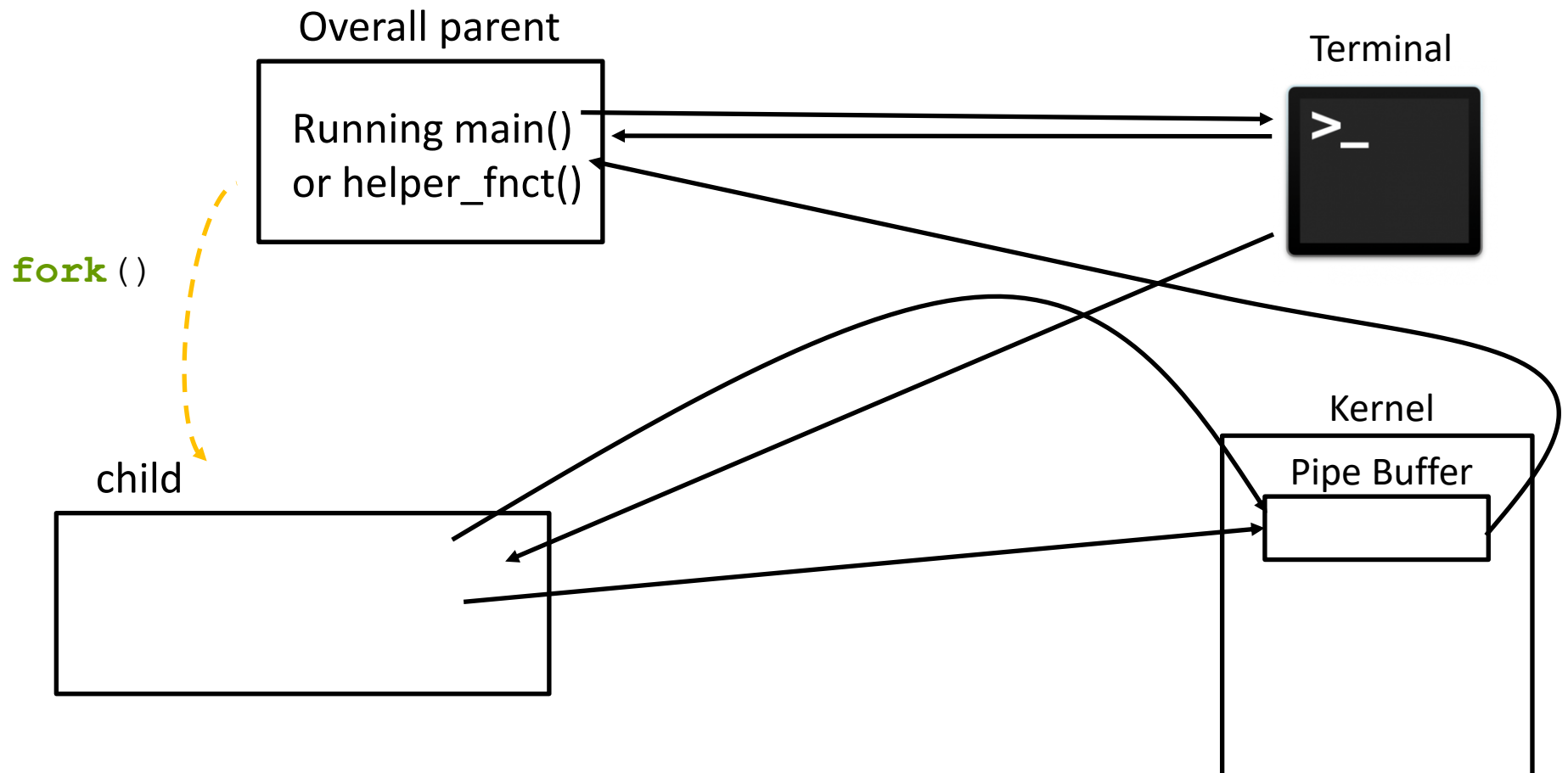
pipe_shell Example Line 1

- Consider the case when a user inputs
  - `"ls | wc"`



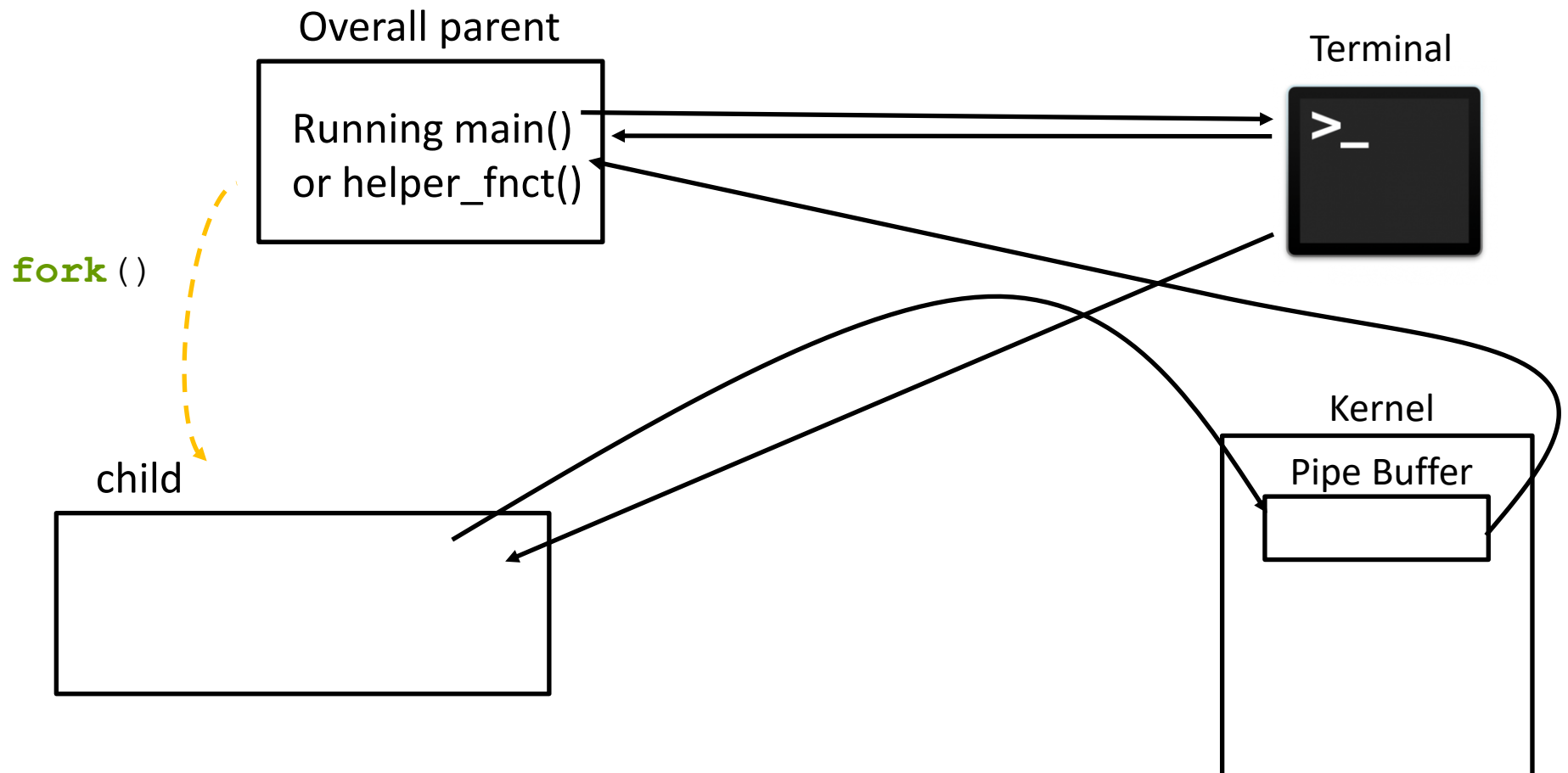
pipe_shell Example Line 1

- Consider the case when a user inputs
  - `"ls | wc"`



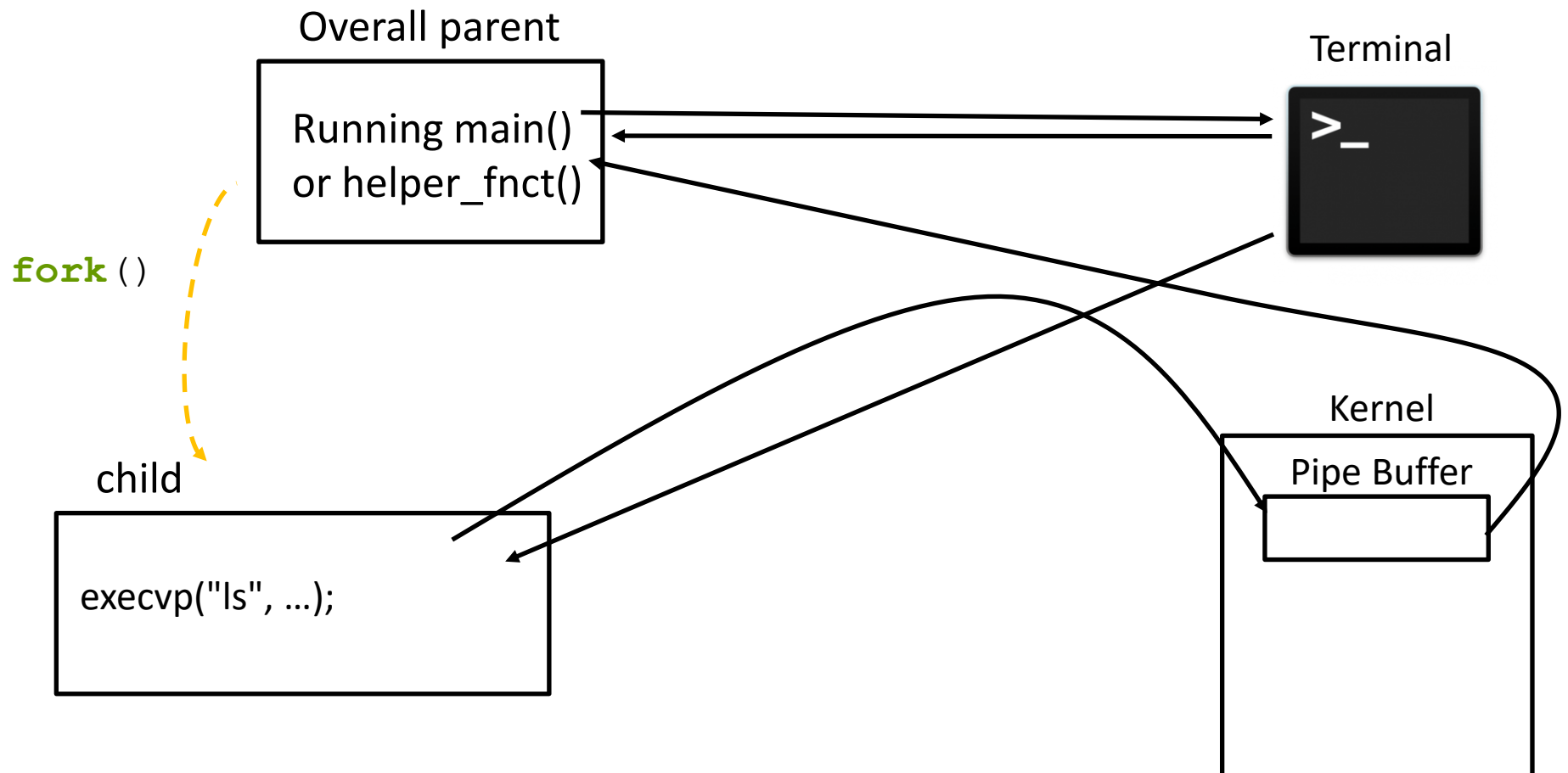
pipe_shell Example Line 1

- Consider the case when a user inputs
  - `"ls | wc"`



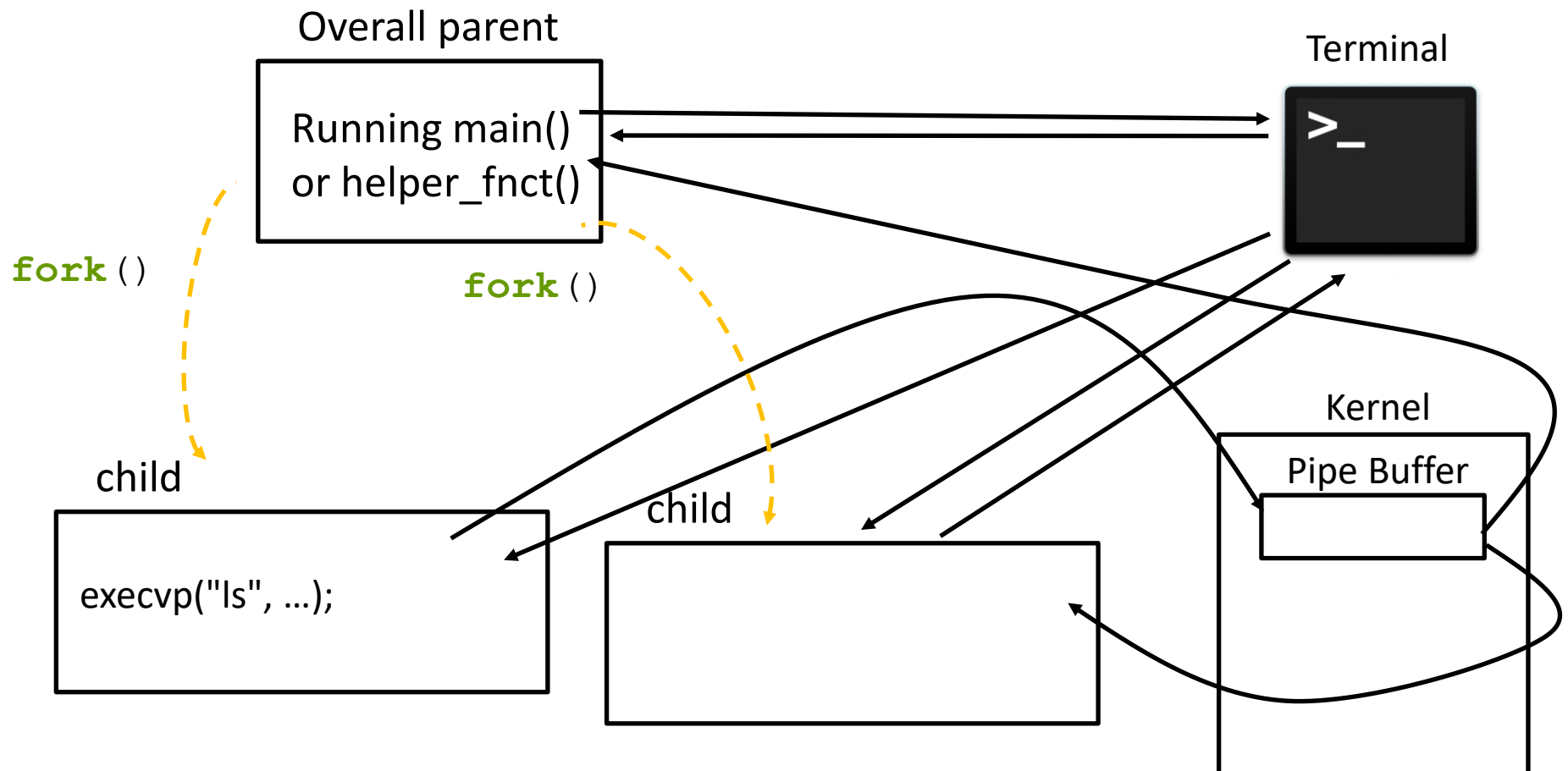
pipe_shell Example Line 1

- Consider the case when a user inputs
  - `"ls | wc"`



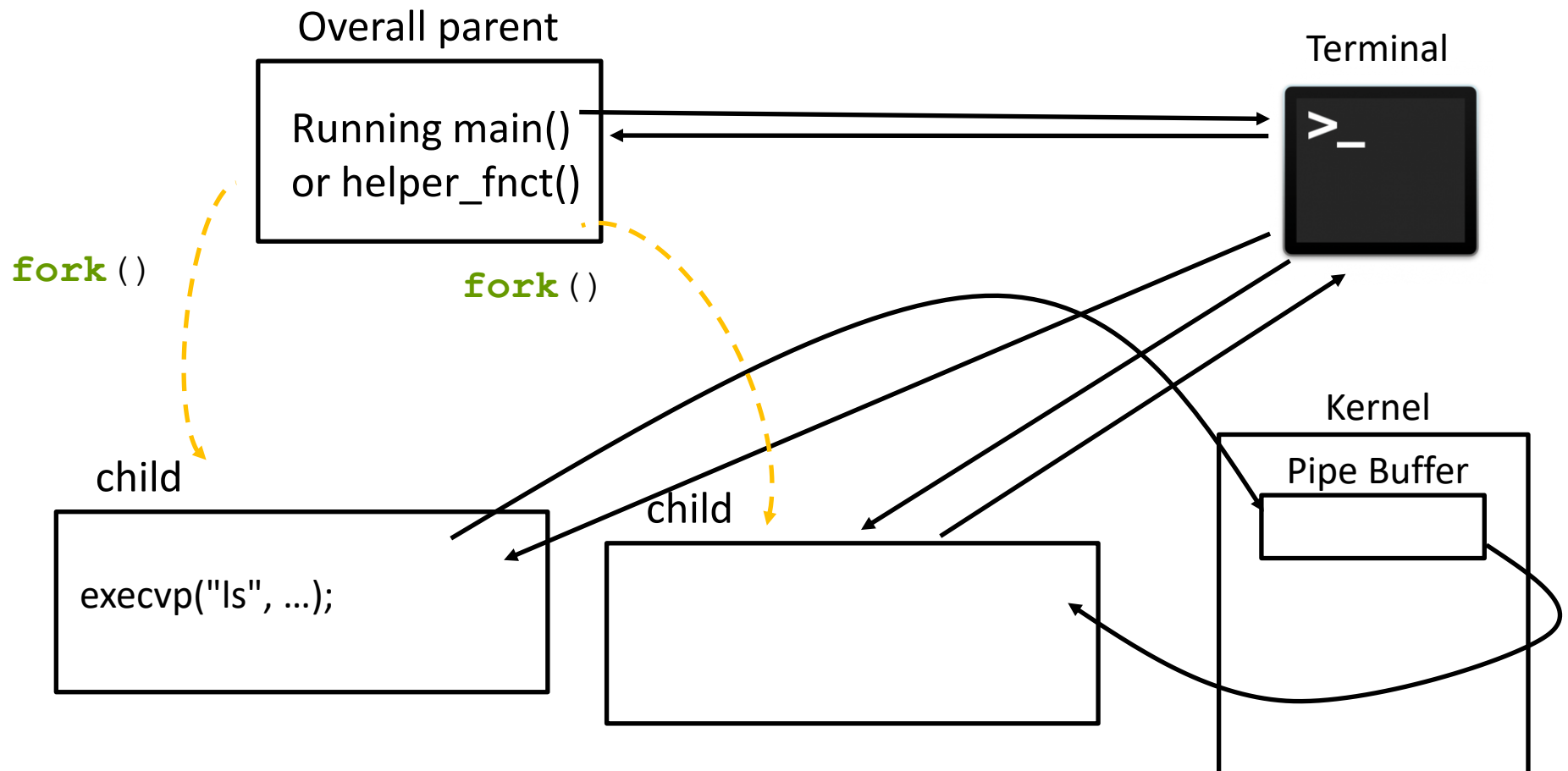
pipe_shell Example Line 1

- Consider the case when a user inputs
  - `"ls | wc"`



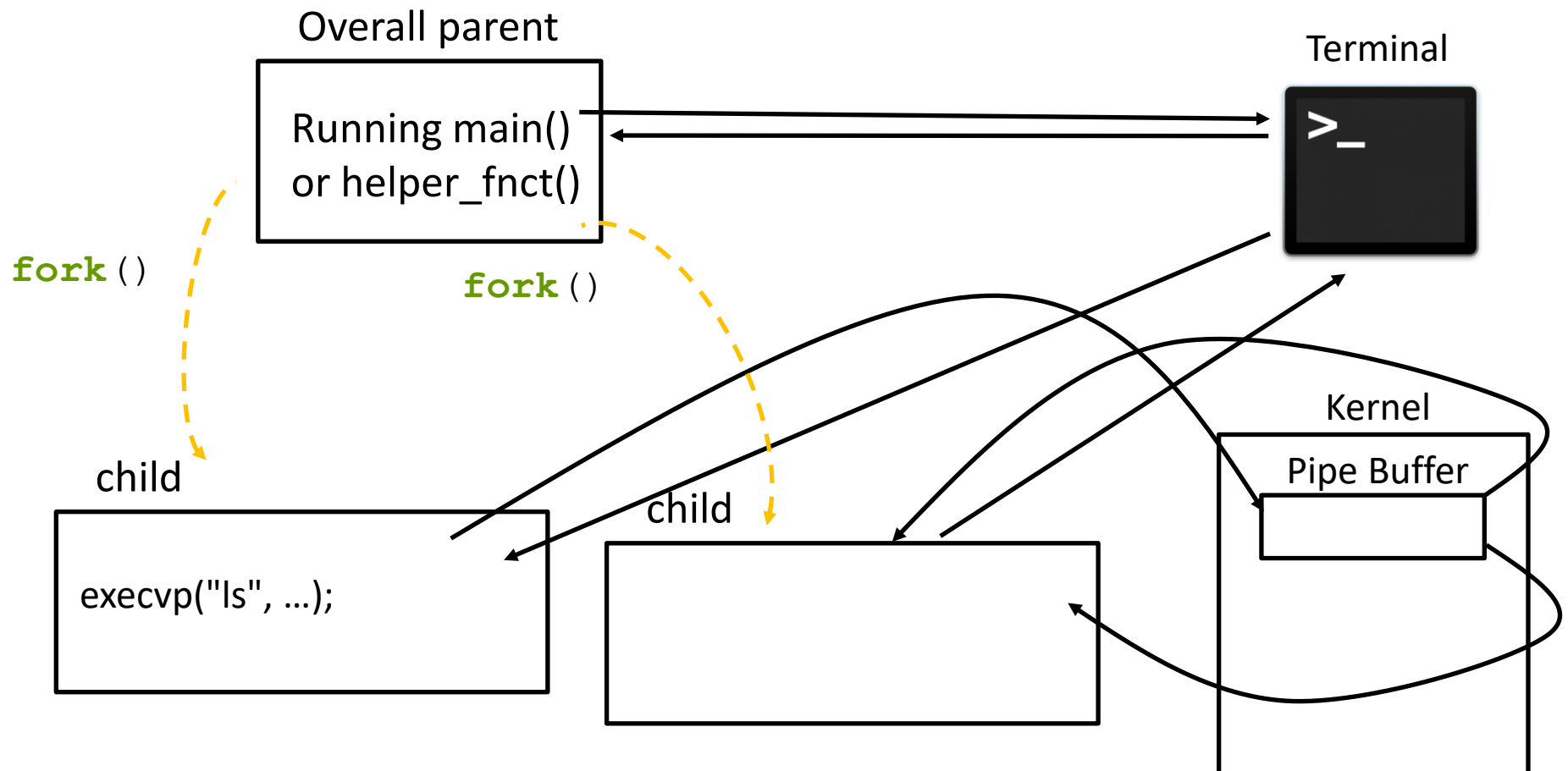
pipe_shell Example Line 1

- Consider the case when a user inputs
  - `"ls | wc"`



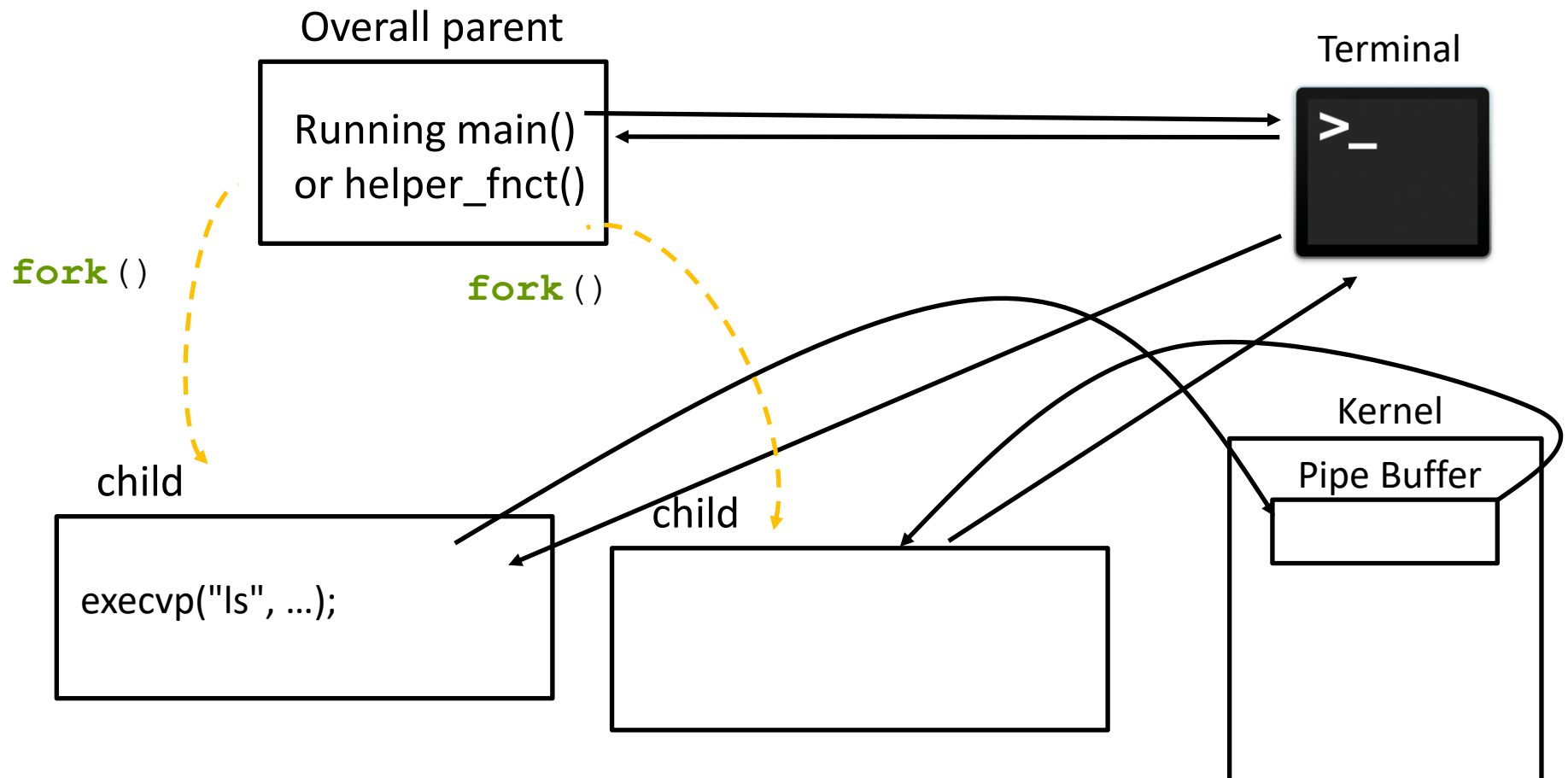
pipe_shell Example Line 1

- Consider the case when a user inputs
  - `"ls | wc"`



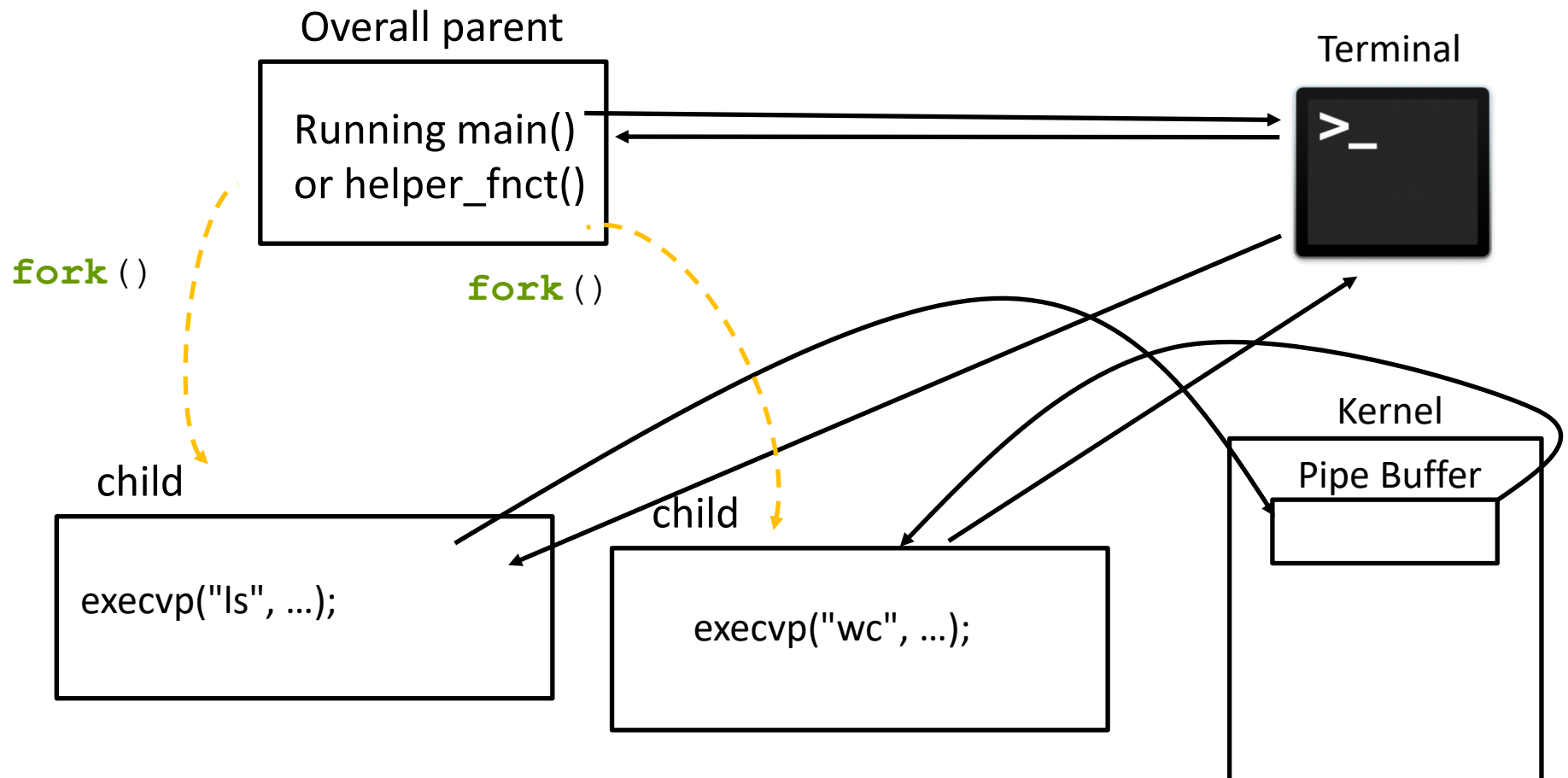
pipe_shell Example Line 1

- Consider the case when a user inputs
  - `"ls | wc"`



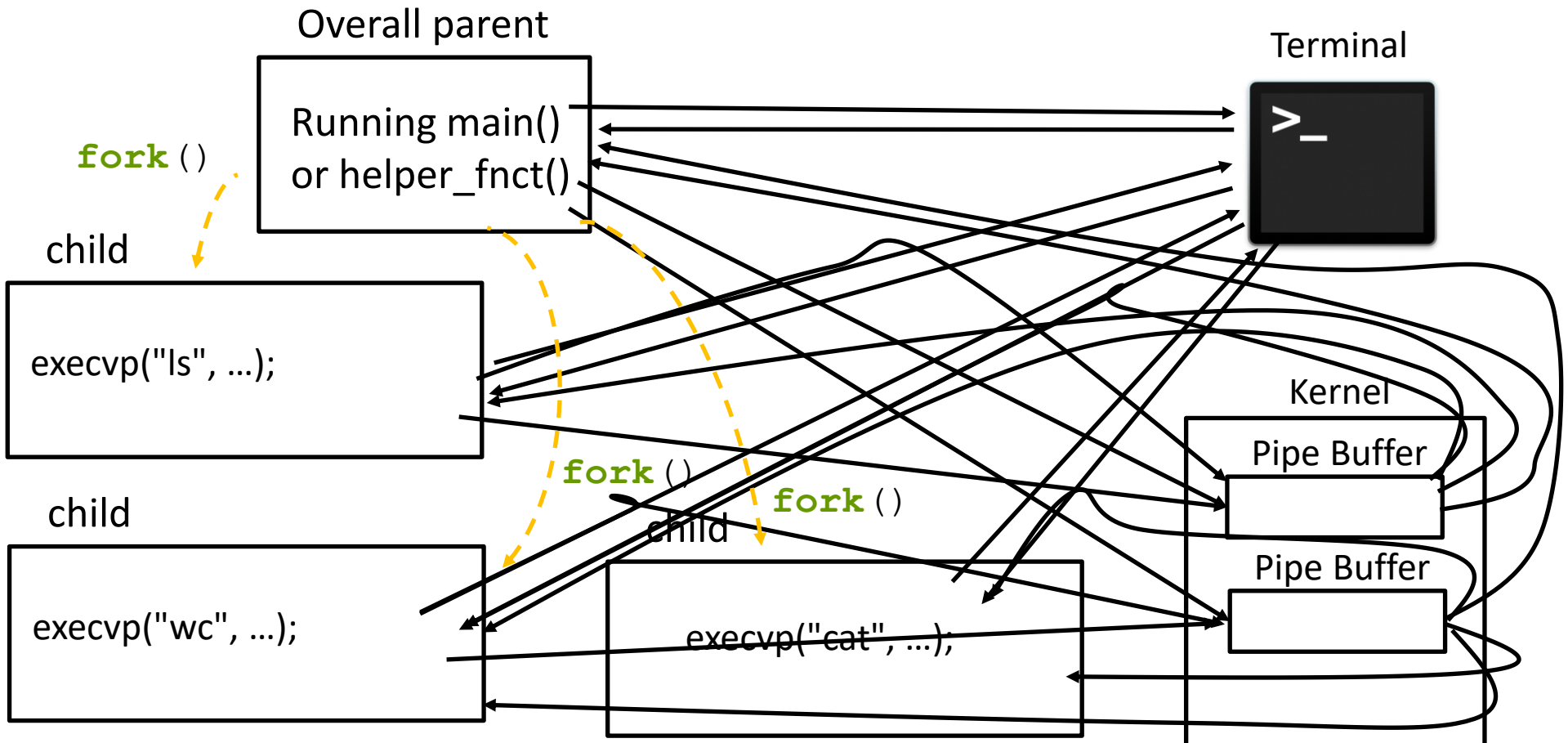
pipe_shell Example Line 1

- Consider the case when a user inputs
  - `"ls | wc"`



pipe_shell Example Line 2

- Consider the case when a user inputs
  - `"ls | wc | cat"`



Suggested “Readings”:

- Take a look at the practice in recitation
- Animation on previous slide available in two_pipe_animation.pptx
- A piece of code that does something similar to the animation can be found in two_pipes.cpp

That's it for now!

- See you next lecture ☺