Pipes

Computer Operating Systems, Fall 2023

Instructor: Travis McGaha

Head TAs: Nate Hoagland & Seungmin Han

TAs:

Andy Jiang	Haoyun Qin	Kevin Bernat	Ryoma Harris
Audrey Yang	Jason hom	Leon Hertzberg	Shyam Mehta
August Fu	Jeff Yang	Maxi Liu	Tina Kokoshvili
Daniel Da	Jerry Wang	Ria Sharma	Zhiyan Lu
Ernest Ng	Jinghao Zhang	Rohan Verma	

pollev.com/tqm

- How is proj0 going?

Administrivia

- Proj0 (penn-shredder) Due **TOMORROW** @ 11:59 pm
  - This includes git & docker setup instructions. Do this part ASAP, it can take a while to debug issues with setup
  - Some people haven't started or are building on local mac

PLEASE SET THINGS UP

  - **This assignment is done on your own**
  - **Late days still exist though (and they are applied automatically)**

Administrivia

- Recitation 2 after lecture today 3:30 – 4:30 in Moore 100C
  - Going over Signals (for proj0, not the stuff I introduce today)
  - May also do some GDB & Terminal Commands?
  - Leave some time for open OH for proj0

- I have OH today Friday 4pm – 7pm, Levine 269A
  - Tentative 4:30-6:30 on Tuesdays for future weeks
  - Longer this time due to proj0 being due

- Peer Evaluation & Project1 to be released later this week
  - **Find a partner and sign up in a group on canvas**
  - Decent indicator of good partner for a pair: similar work ethic
- Check-in Quiz 2 Due in ~1 week

pollev.com/tqm

- Any questions, comments or concerns from last lecture?

Lecture Outline

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

Pipes

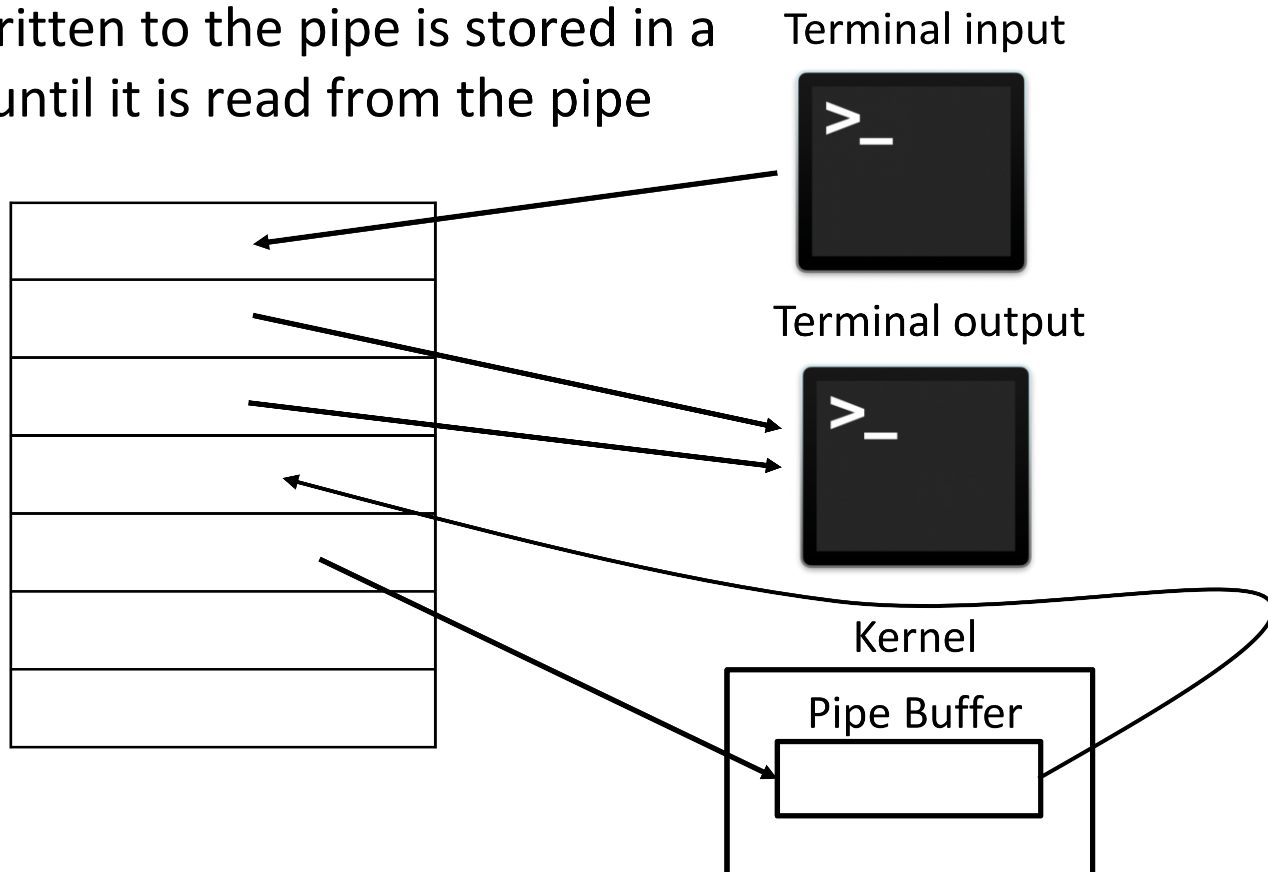
```
int pipe(int pipefd[2]);
```

- 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



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**

pollev.com/tqm

- What does this program do? (assume no system calls fail)

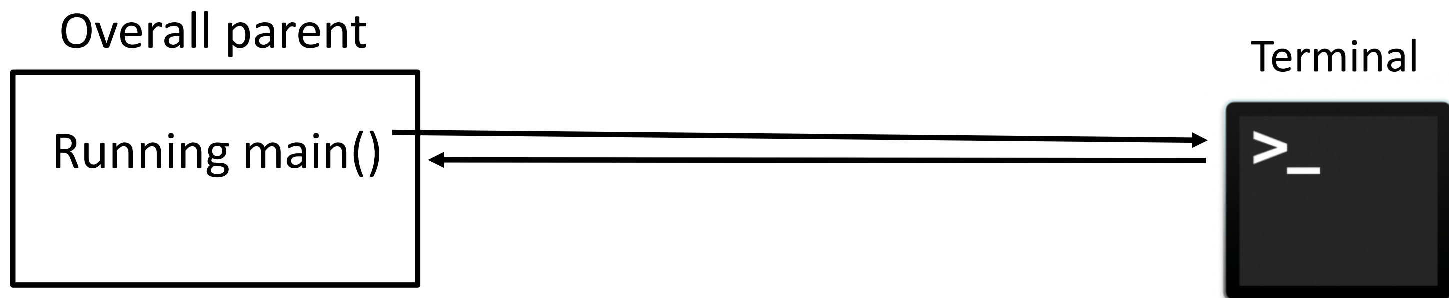
```
12 int main() {
13   // Note: it is still the parent process here
14   int pipe_fds[2];
15   pipe(pipe_fds);
16
17   // child process only exits after this
18   pid_t pid = fork();
19
20   if (pid == 0) {
21     // child process
22
23     /// close the end of the pipe that isn't used
24     close(pipe_fds[1]);
25     dup2(pipe_fds[0], STDIN_FILENO);
26     close(pipe_fds[0]);
27
28     char buf[BUF_SIZE + 1];
29
30     ssize_t chars_read = read(STDIN_FILENO, buf, BUF_SIZE);
31     while(chars_read > 0) {
32       buf[chars_read] = '\0';
33       printf("%s", buf);
34       chars_read = read(STDIN_FILENO, buf, BUF_SIZE);
35     }
36
37     exit(EXIT_SUCCESS);
38   }
39   // parent
40
```

code is on website as `cat_pipe.c`

```
39   // parent
40
41   /// close the end of the pipe I won't use
42   close(pipe_fds[0]);
43
44   int fd = open("mutual_aid.txt", O_RDONLY);
45
46   char buf[BUF_SIZE];
47   ssize_t chars_read = read(fd ,buf, BUF_SIZE);
48   while(chars_read > 0) {
49     write(pipe_fds[1], buf, chars_read);
50     chars_read = read(fd, buf, BUF_SIZE);
51   }
52
53   int wstatus;
54   wait(&wstatus);
55
56   return EXIT_SUCCESS;
57 }
```

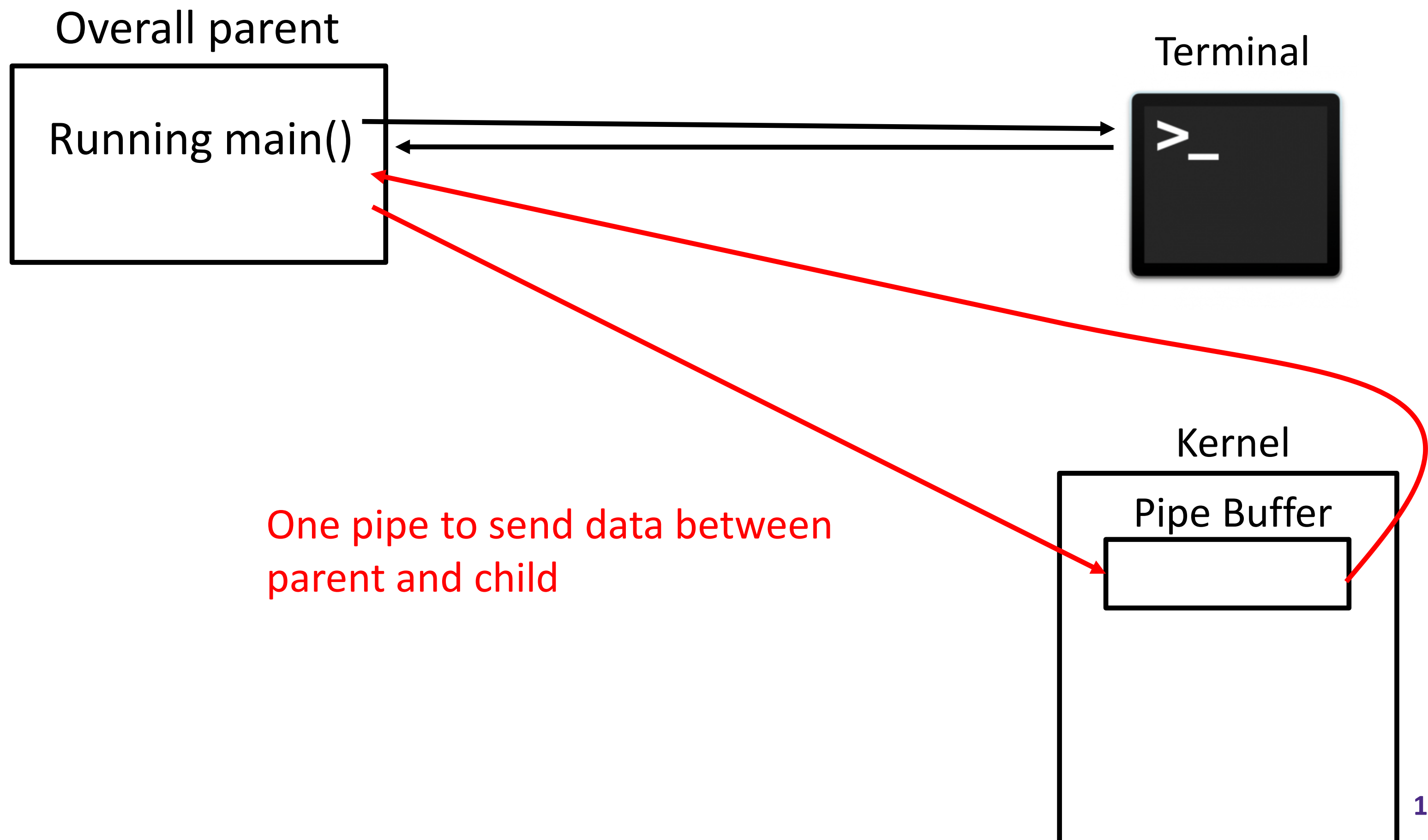
`cat_pipe.c` Trace

- First:
  we create a pipe



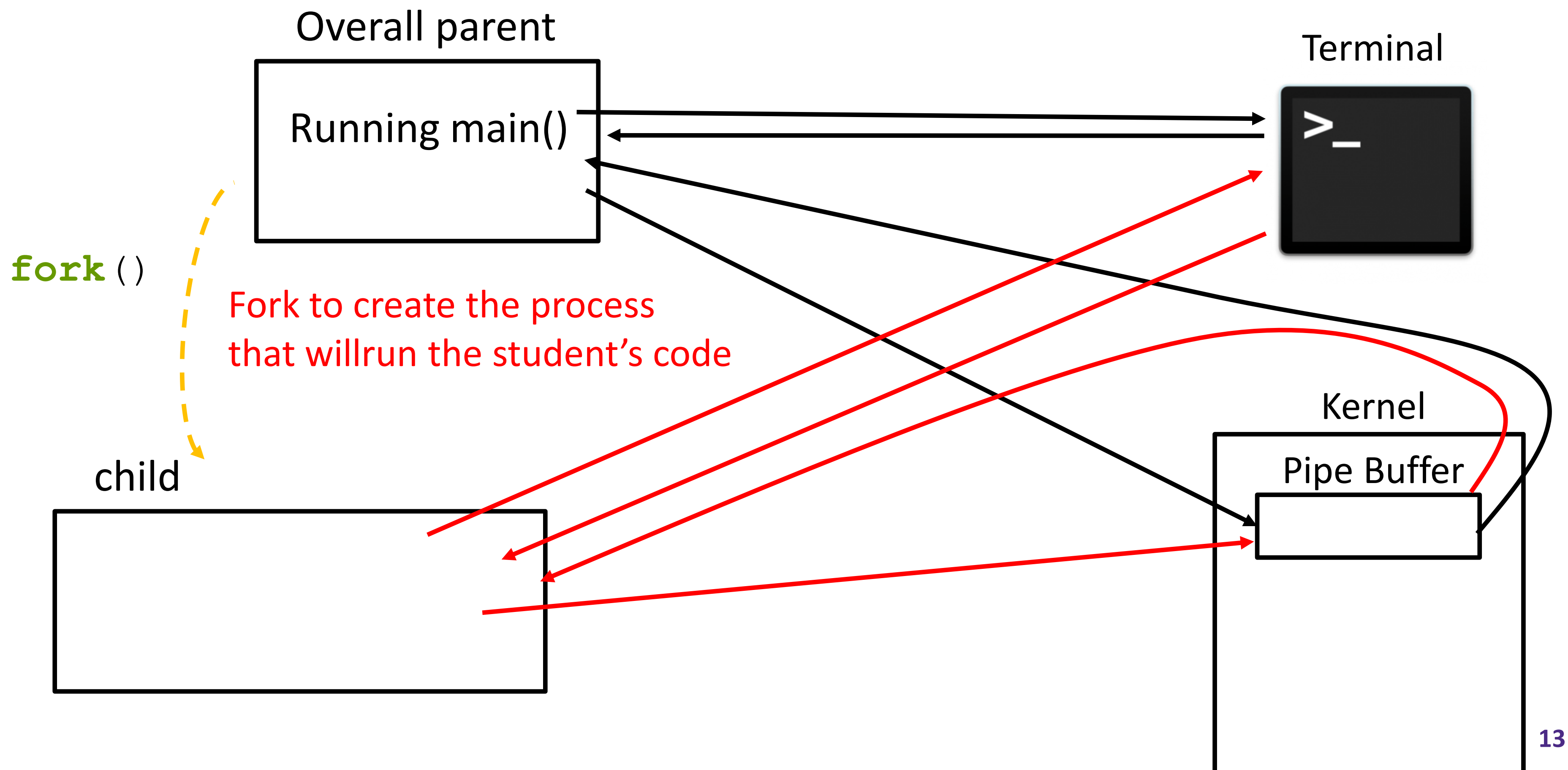
`cat_pipe.c` Trace

- First:
  we create a pipe



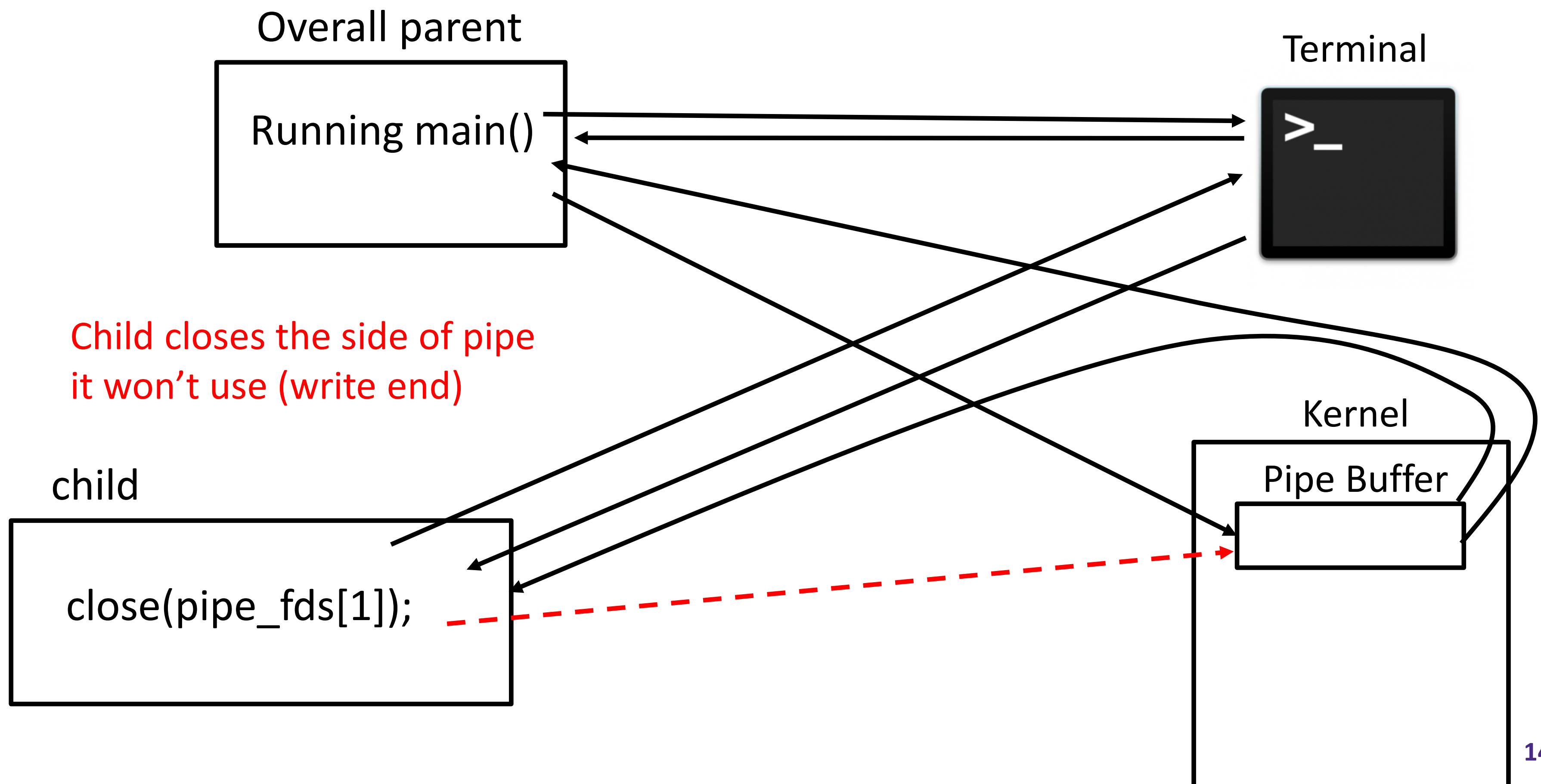
`cat_pipe.c` Trace

- second
  Fork



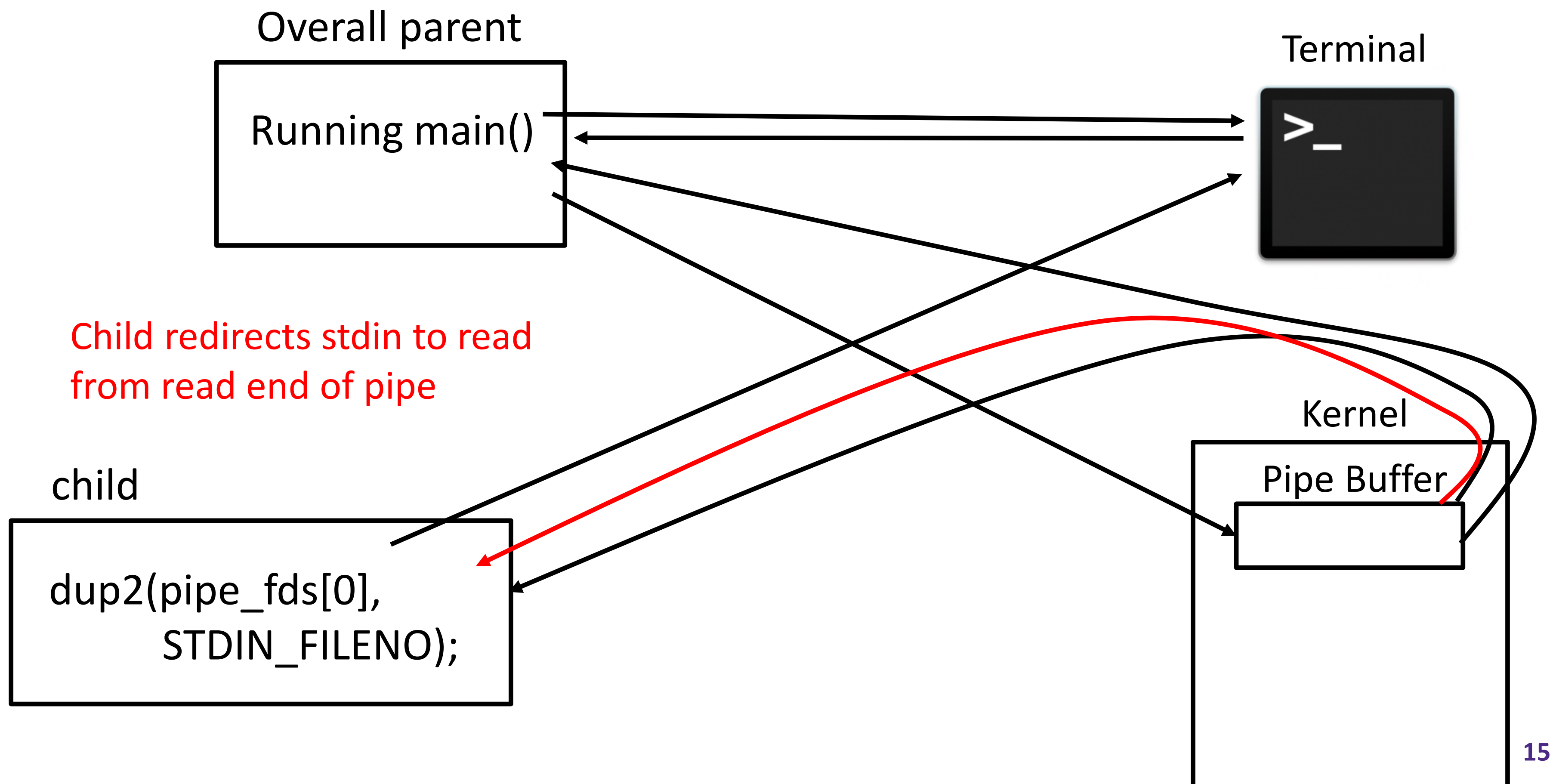
`cat_pipe.c` Trace

- Walking through child, but parent could be running first, after, or at the same time



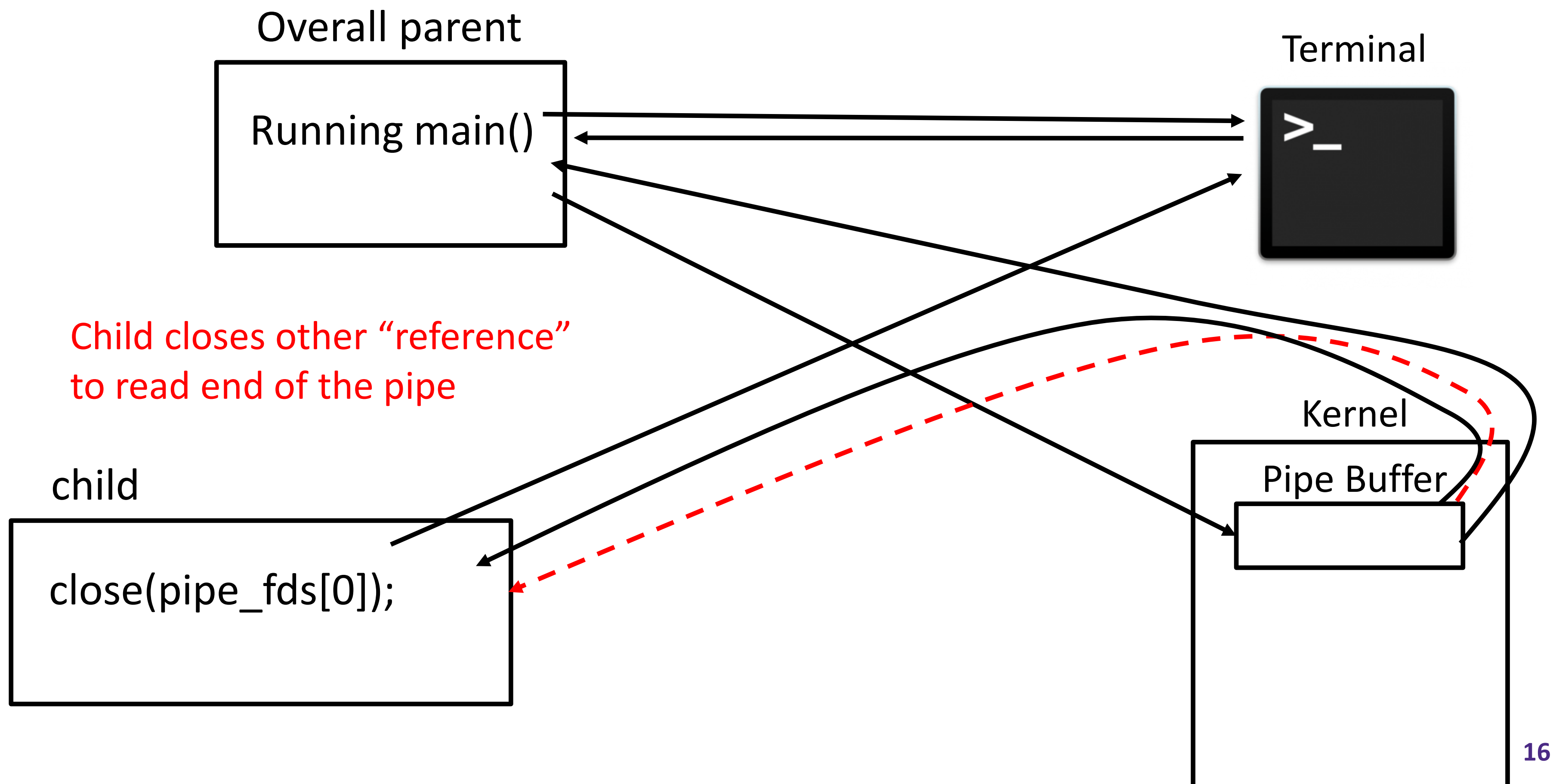
`cat_pipe.c` Trace

- Walking through child, but parent could be running first, after, or at the same time



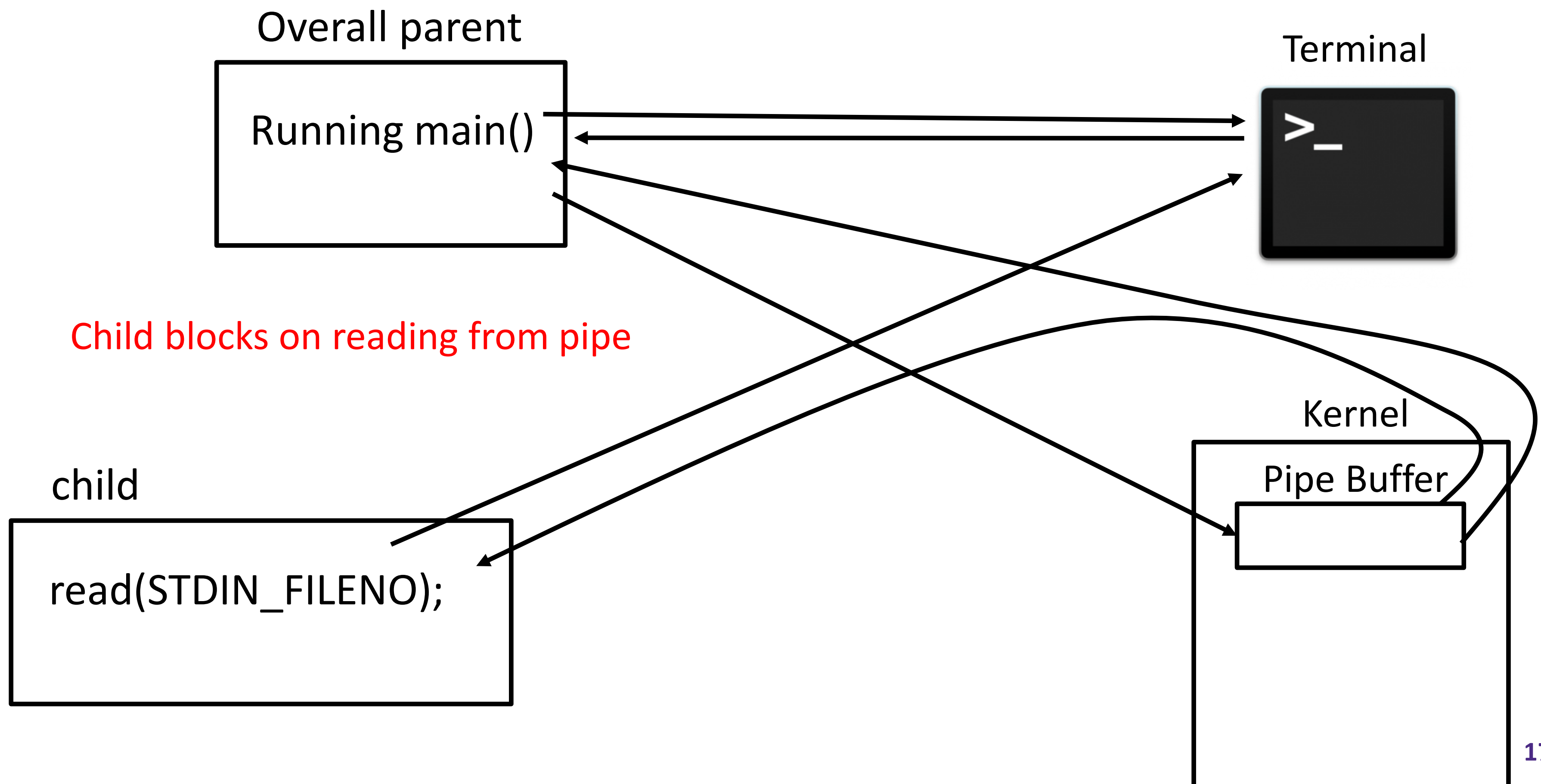
`cat_pipe.c` Trace

- Walking through child, but parent could be running first, after, or at the same time



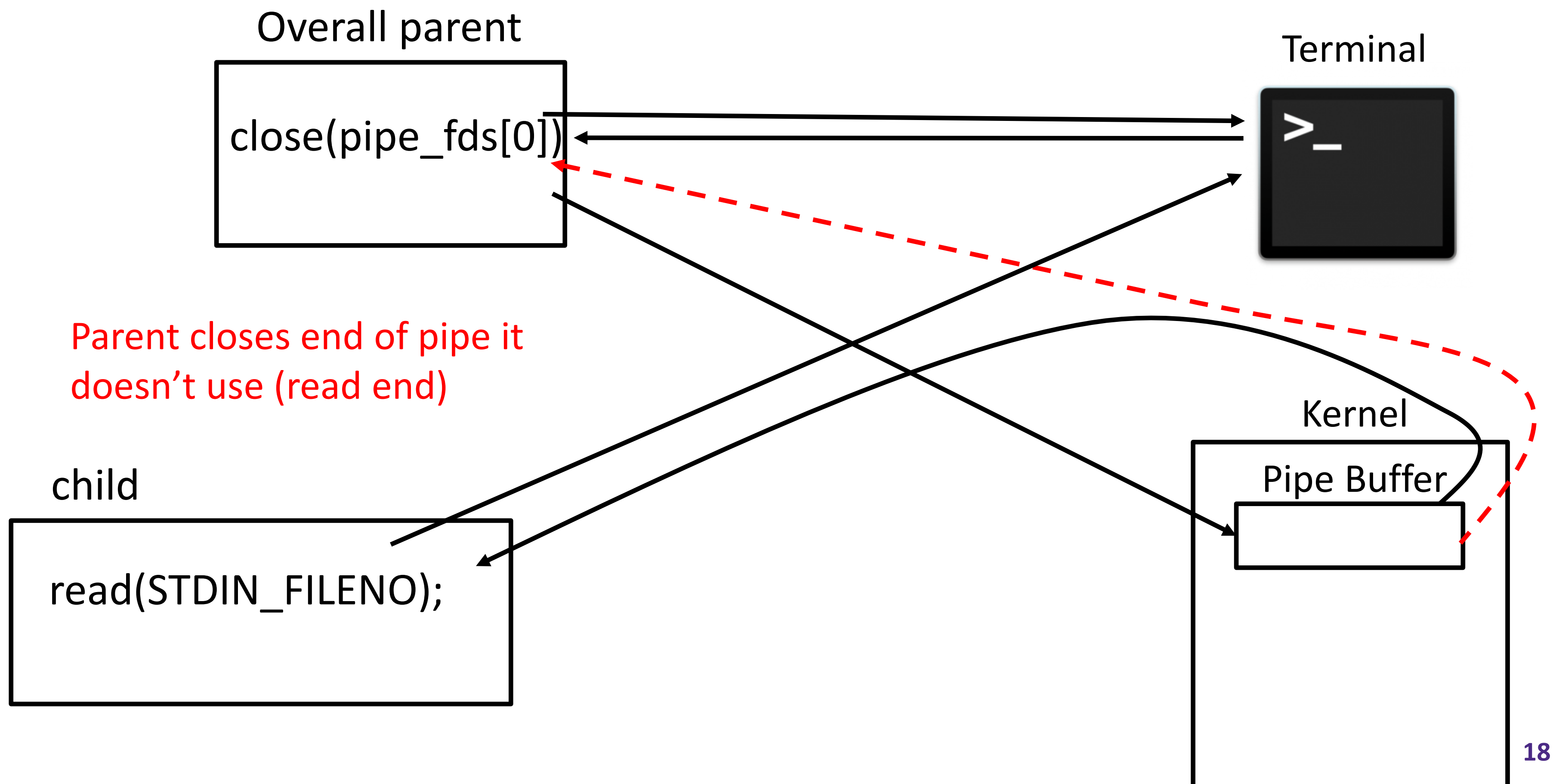
`cat_pipe.c` Trace

- Walking through child, but parent could be running first, after, or at the same time



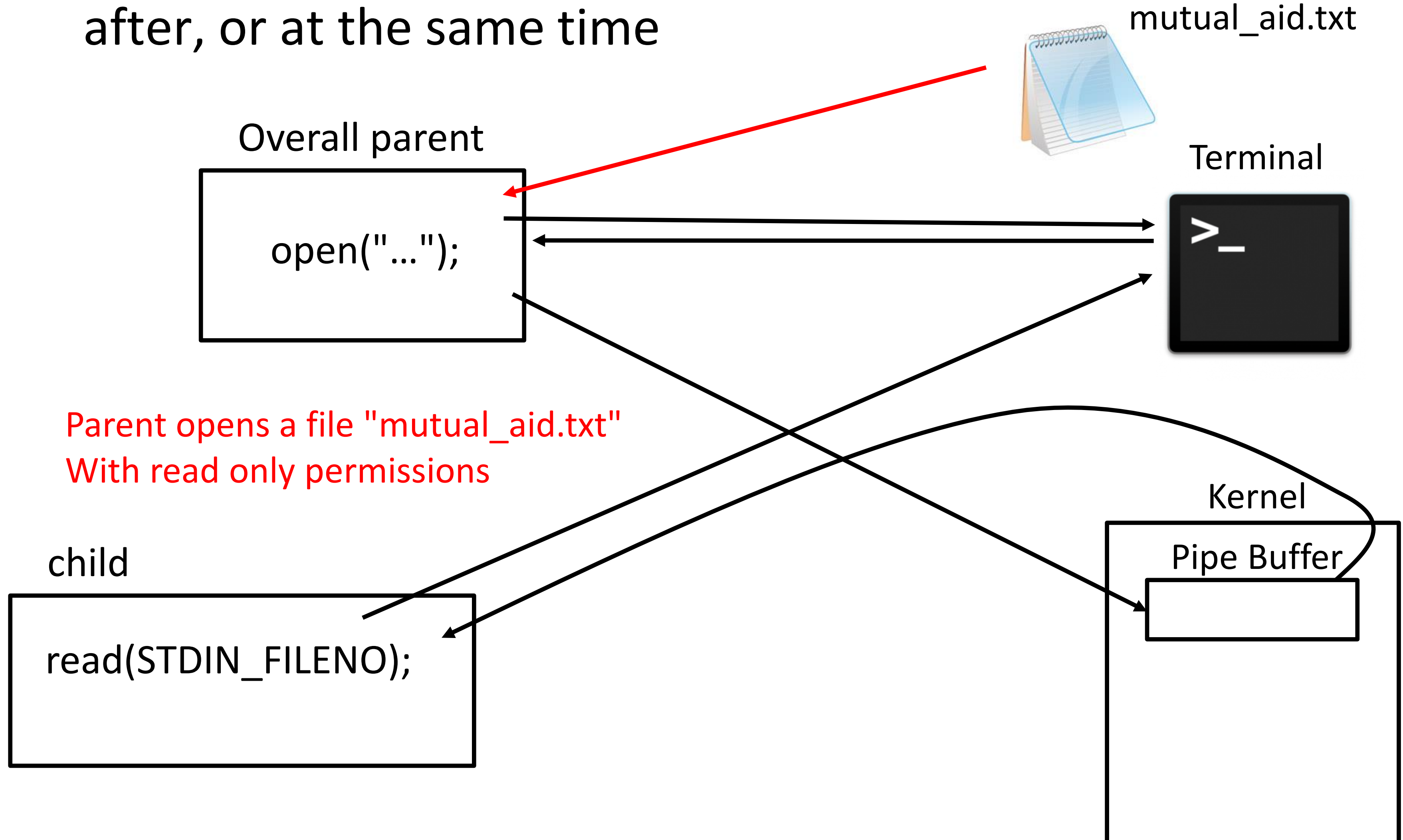
`cat_pipe.c` Trace

- Walking through **parent**, but child could be running first, after, or at the same time



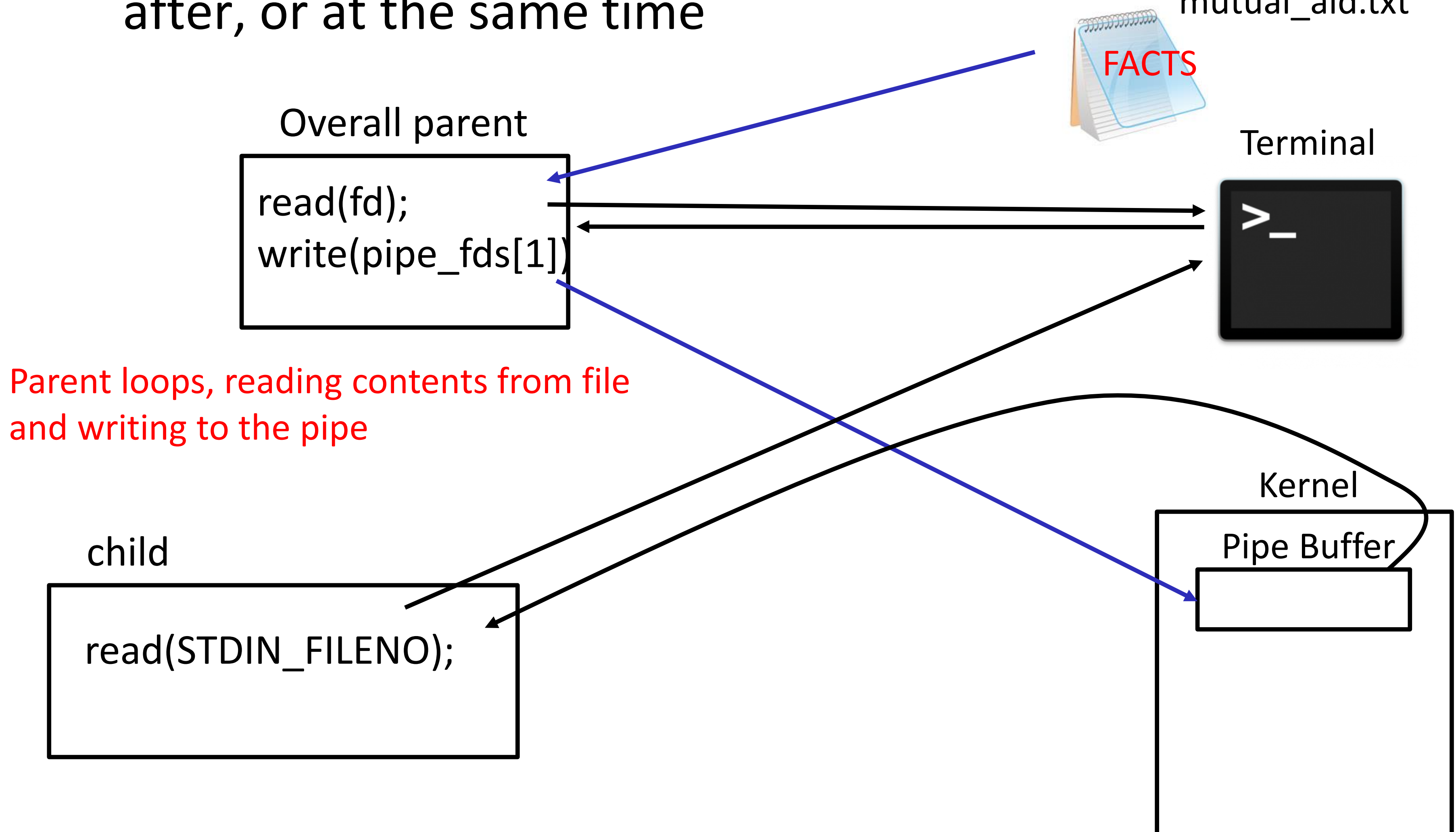
cat_pipe.c Trace

- Walking through **parent**, but child could be running first, after, or at the same time



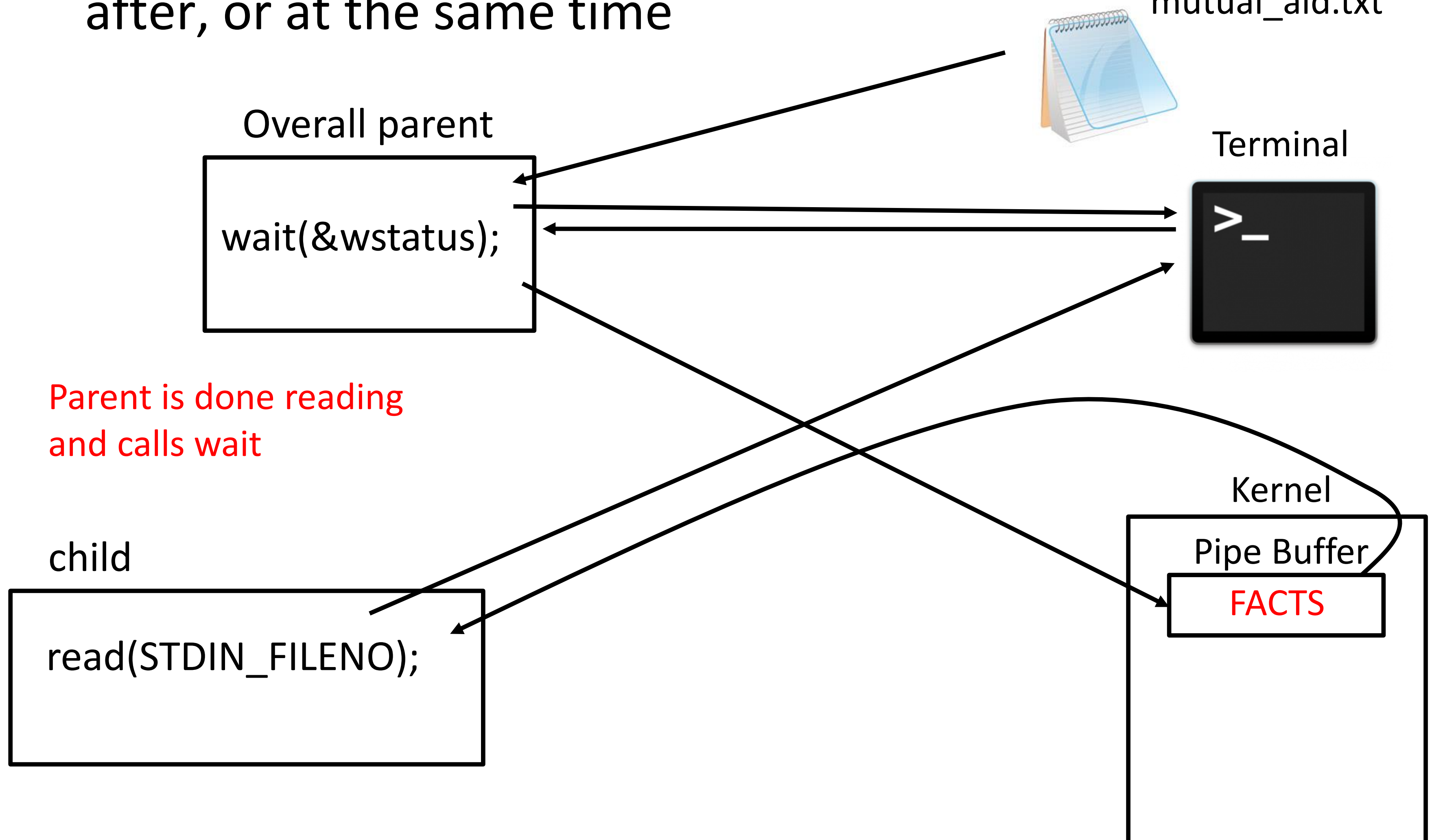
cat_pipe.c Trace

- Walking through **parent**, but child could be running first, after, or at the same time



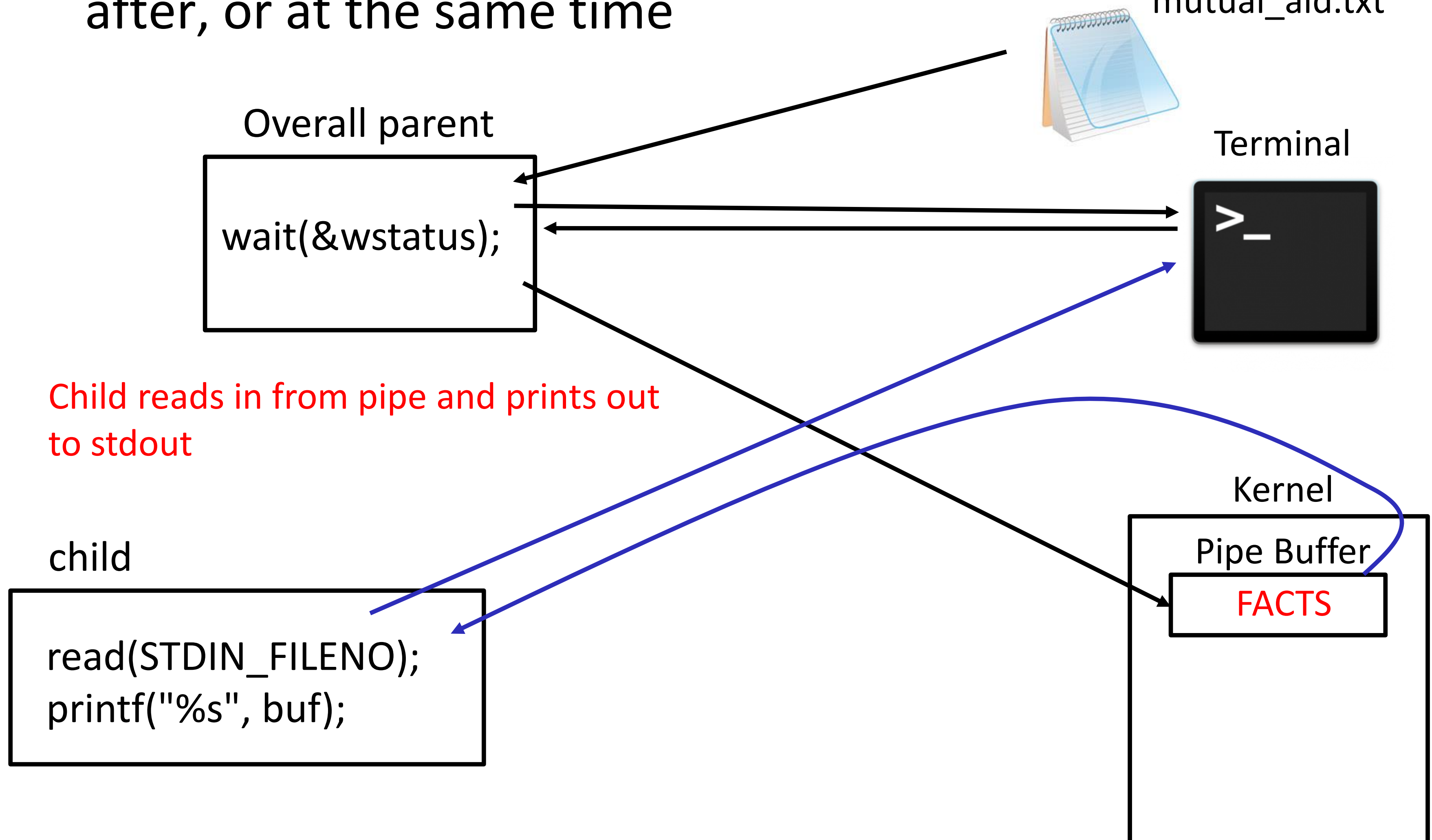
cat_pipe.c Trace

- Walking through **parent**, but child could be running first, after, or at the same time



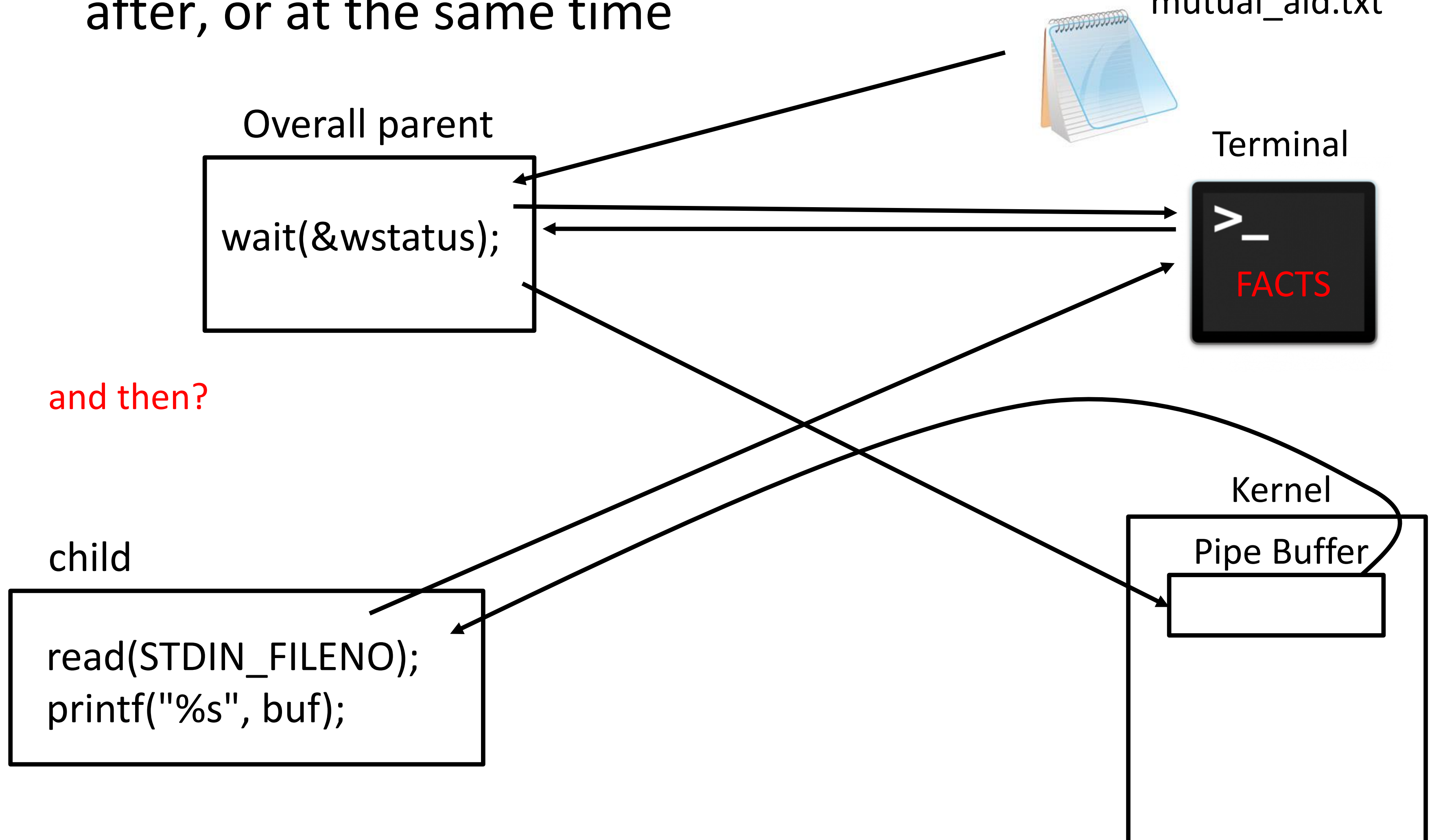
cat_pipe.c Trace

- Walking through **parent**, but child could be running first, after, or at the same time



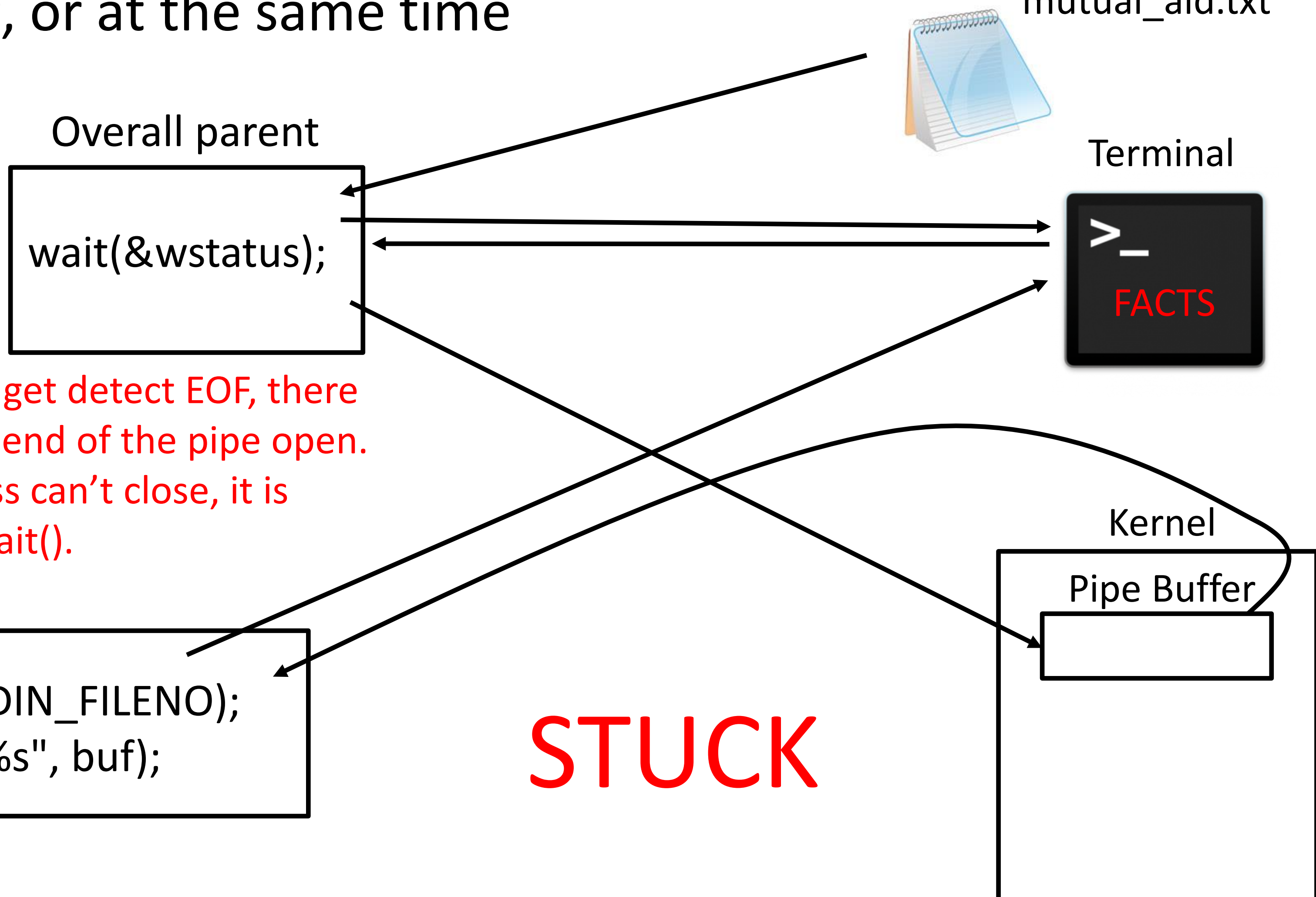
cat_pipe.c Trace

- Walking through **parent**, but child could be running first, after, or at the same time



cat_pipe.c Trace

- Walking through **parent**, but child could be running first, after, or at the same time

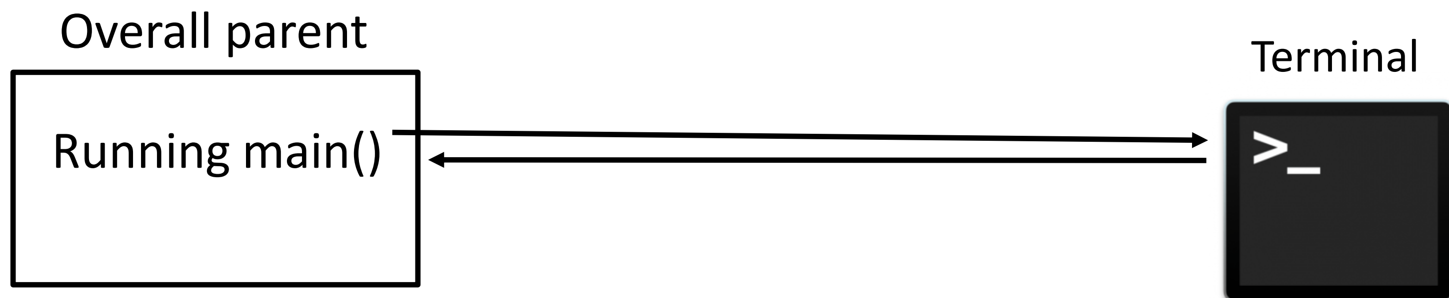


Exec & Pipe Demo

- See `io_autograder.c`
  - How could we take advantage of exec and pipe to do something useful?
  - Combine usage of fork and exec so our program can do multiple things

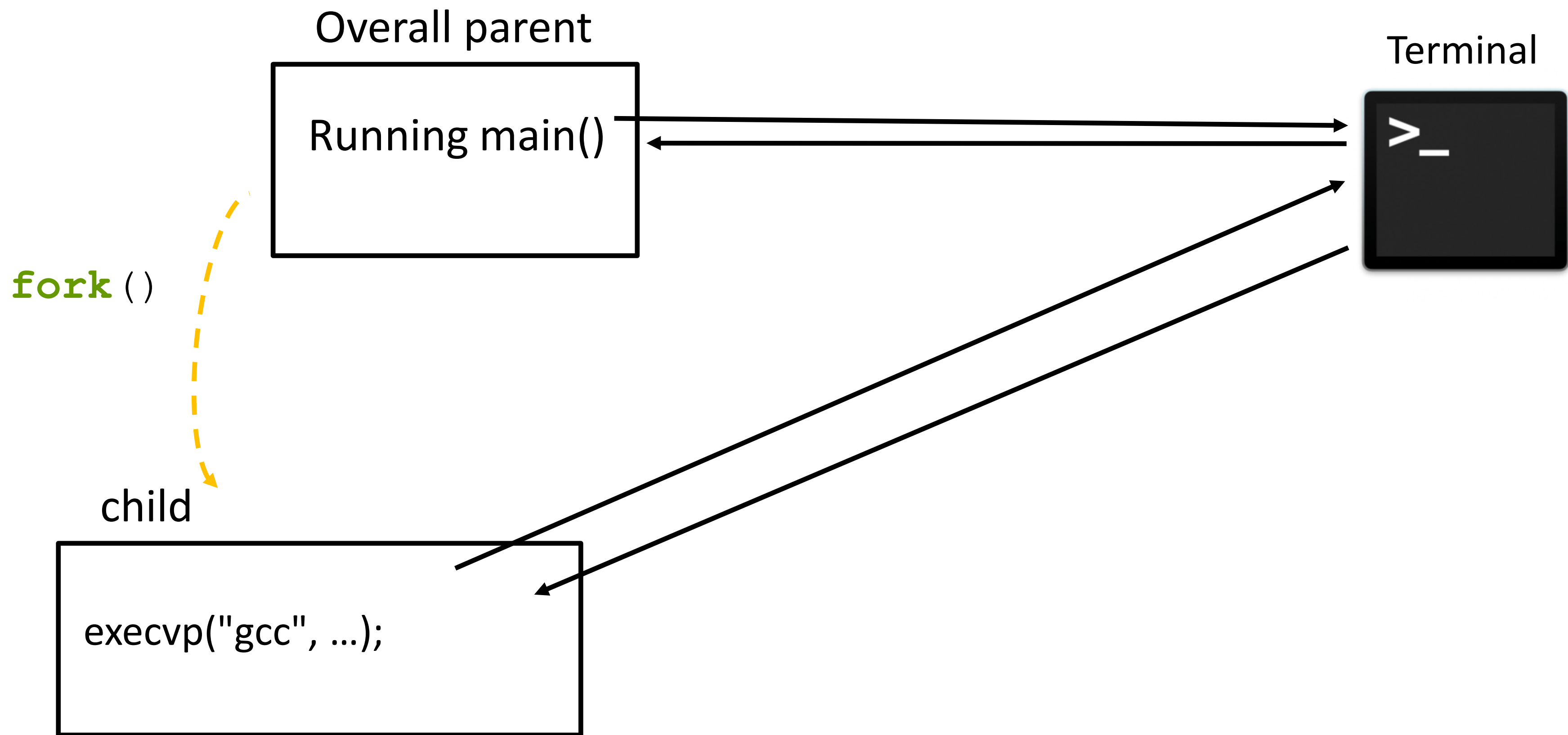
`io_autograder.c` Trace

- First:
  we compile the program with the gcc command



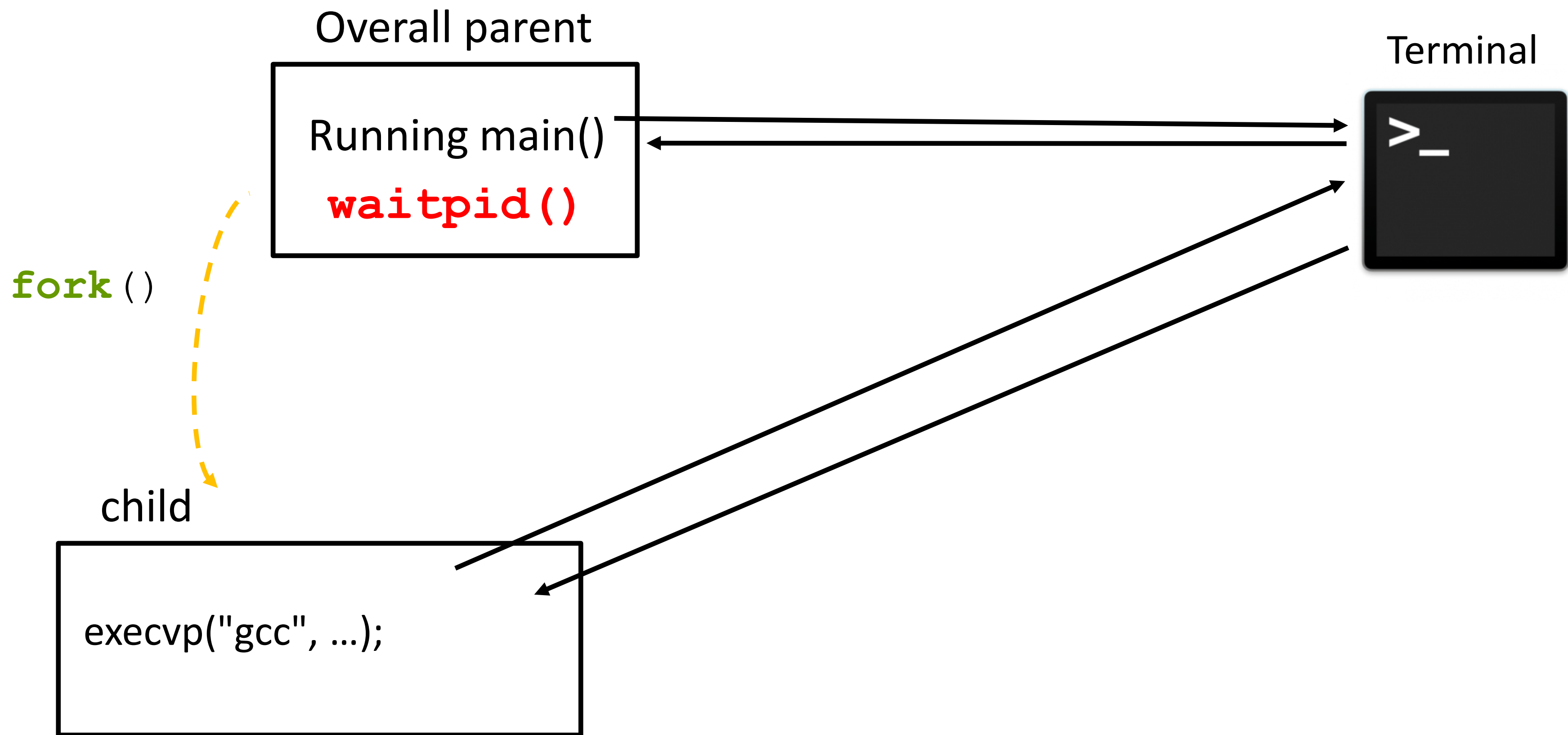
io_autograder.c Trace

- First:
  we compile the program with the gcc command



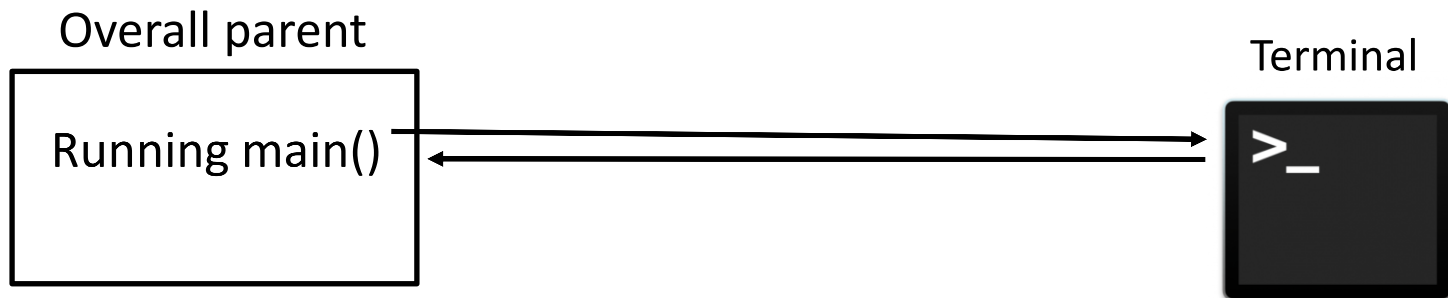
io_autograder.c Trace

- First:
  we compile the program with the gcc command



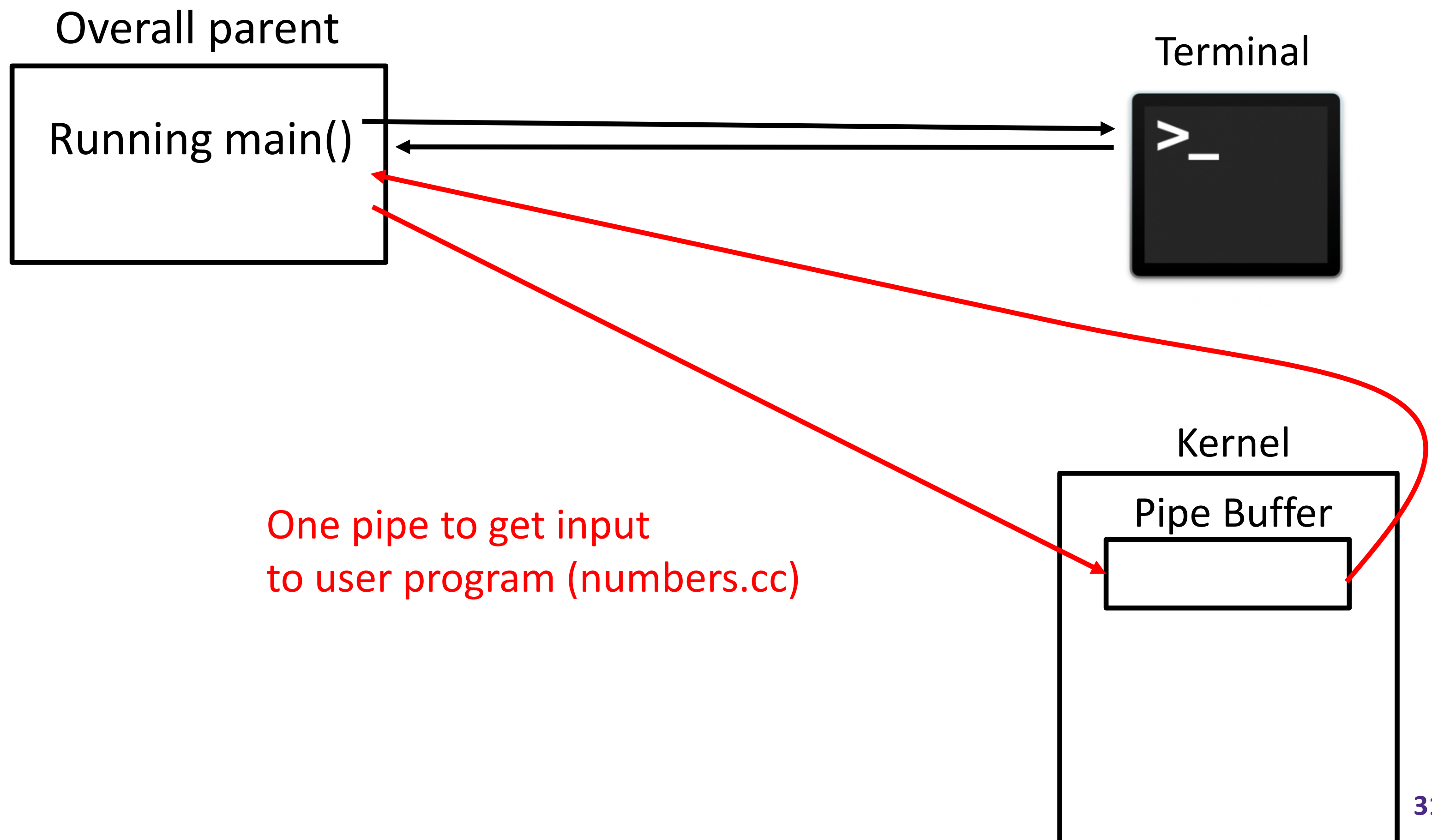
`io_autograder.c` Trace

- Compilation done! Run the compiled program… BUT send autograder input and capture output



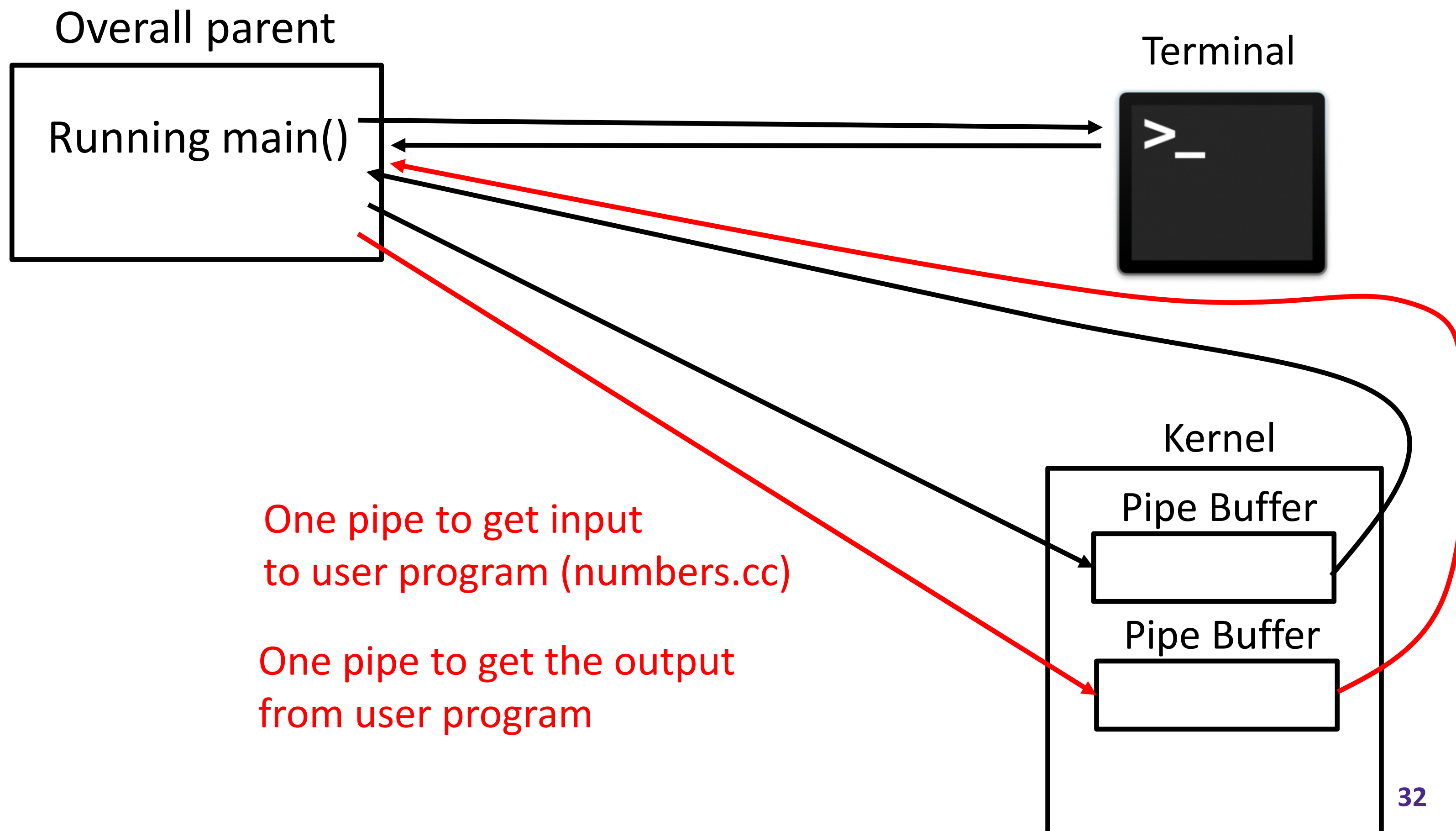
io_autograder.c Trace

- Compilation done! Run the compiled program… BUT send autograder input and capture output



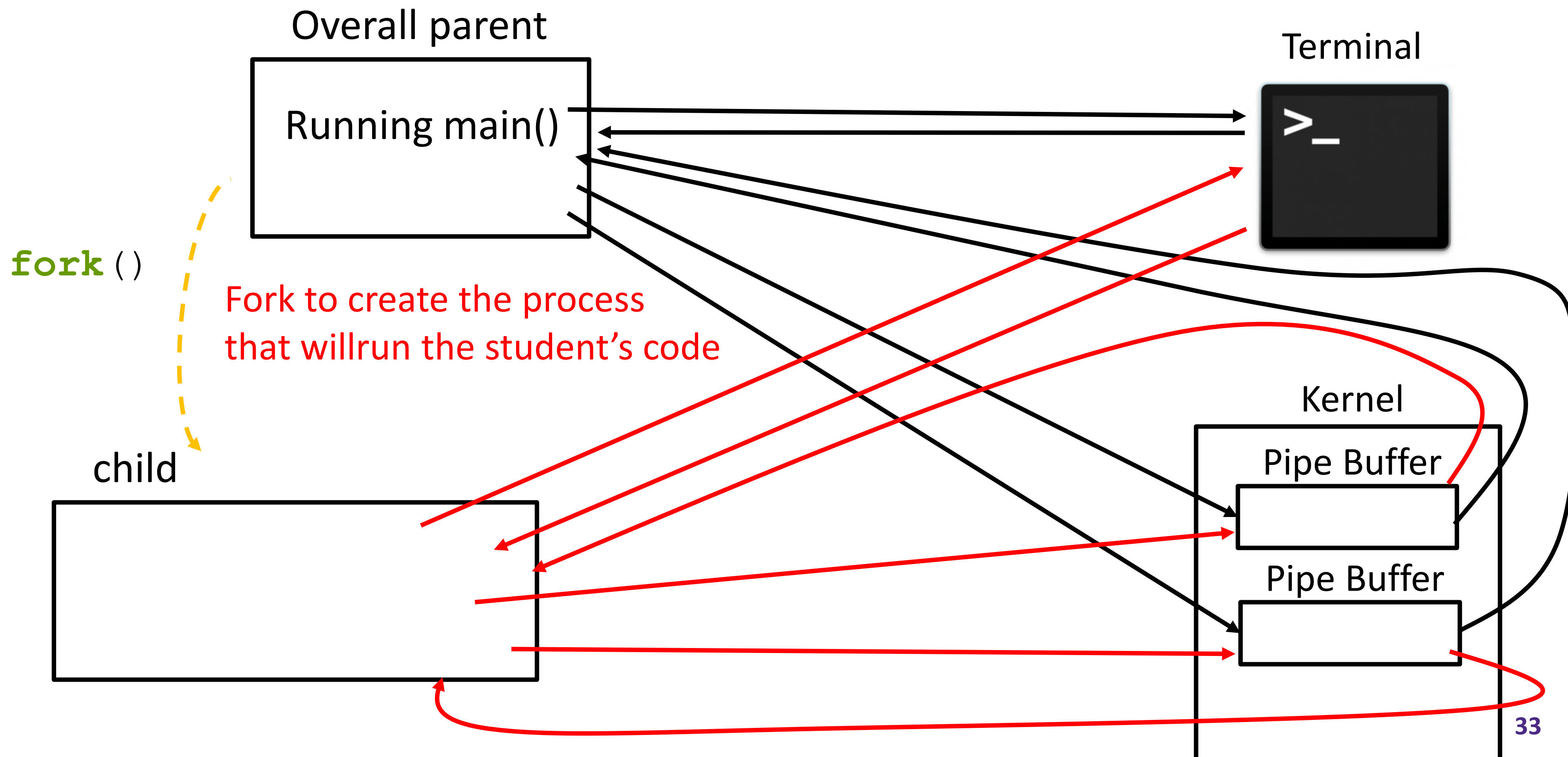
`io_autograder.c` Trace

- Compilation done! Run the compiled program… BUT send autograder input and capture output



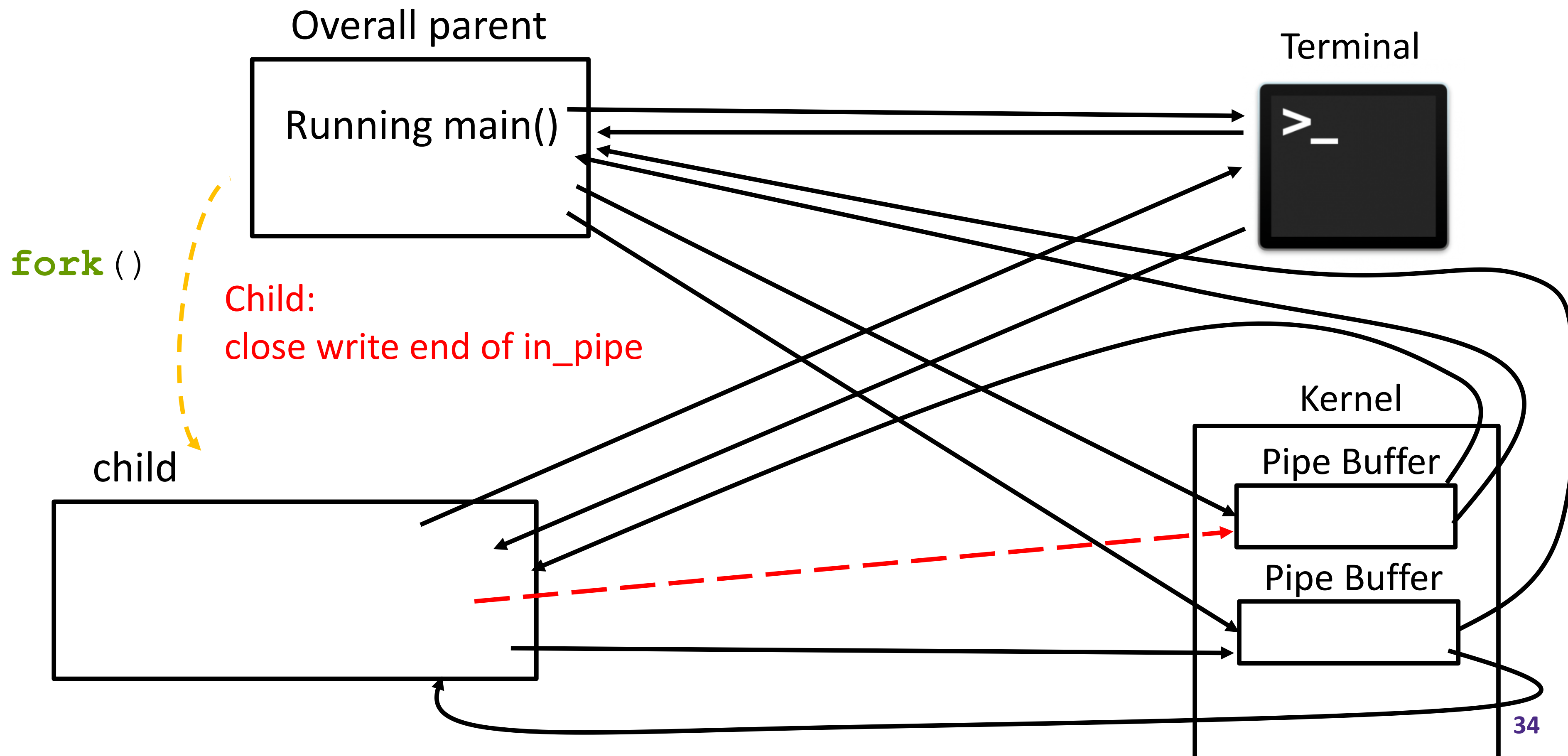
`io_autograder.c` Trace

- Compilation done! Run the compiled program…
  BUT send autograder input and capture output



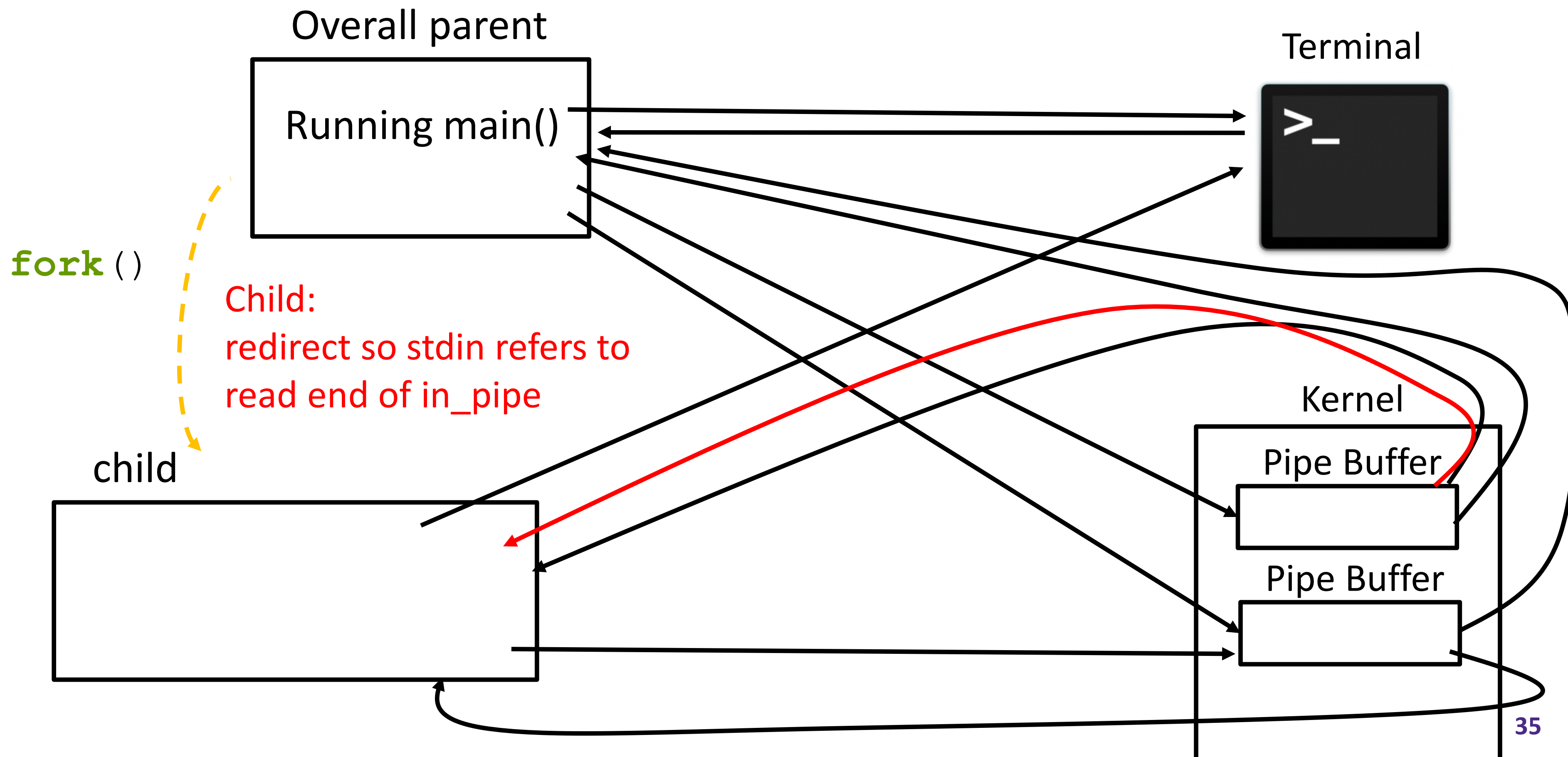
io_autograder.c Trace

- Compilation done! Run the compiled program… BUT send autograder input and capture output



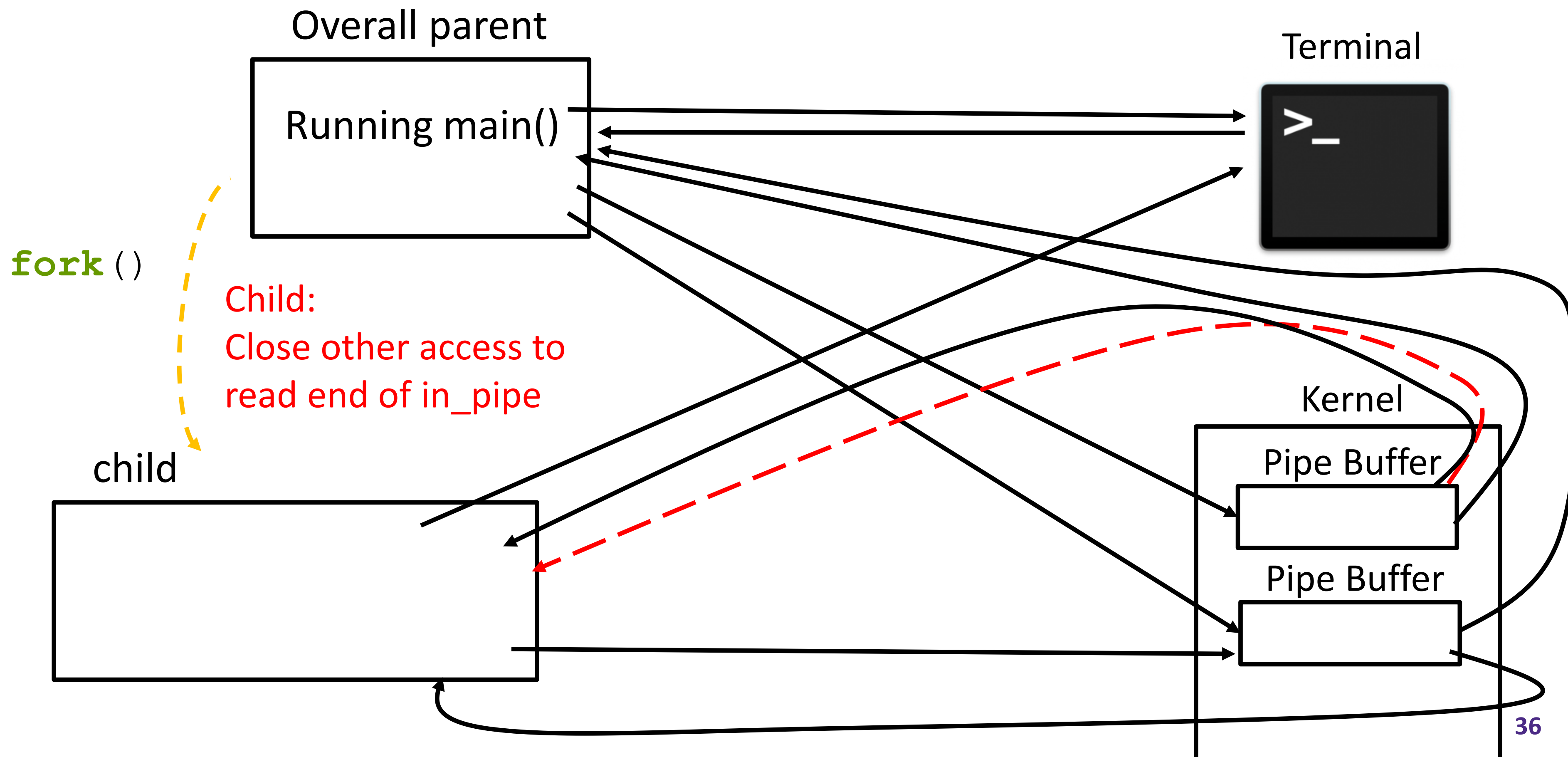
io_autograder.c Trace

- Compilation done! Run the compiled program… BUT send autograder input and capture output



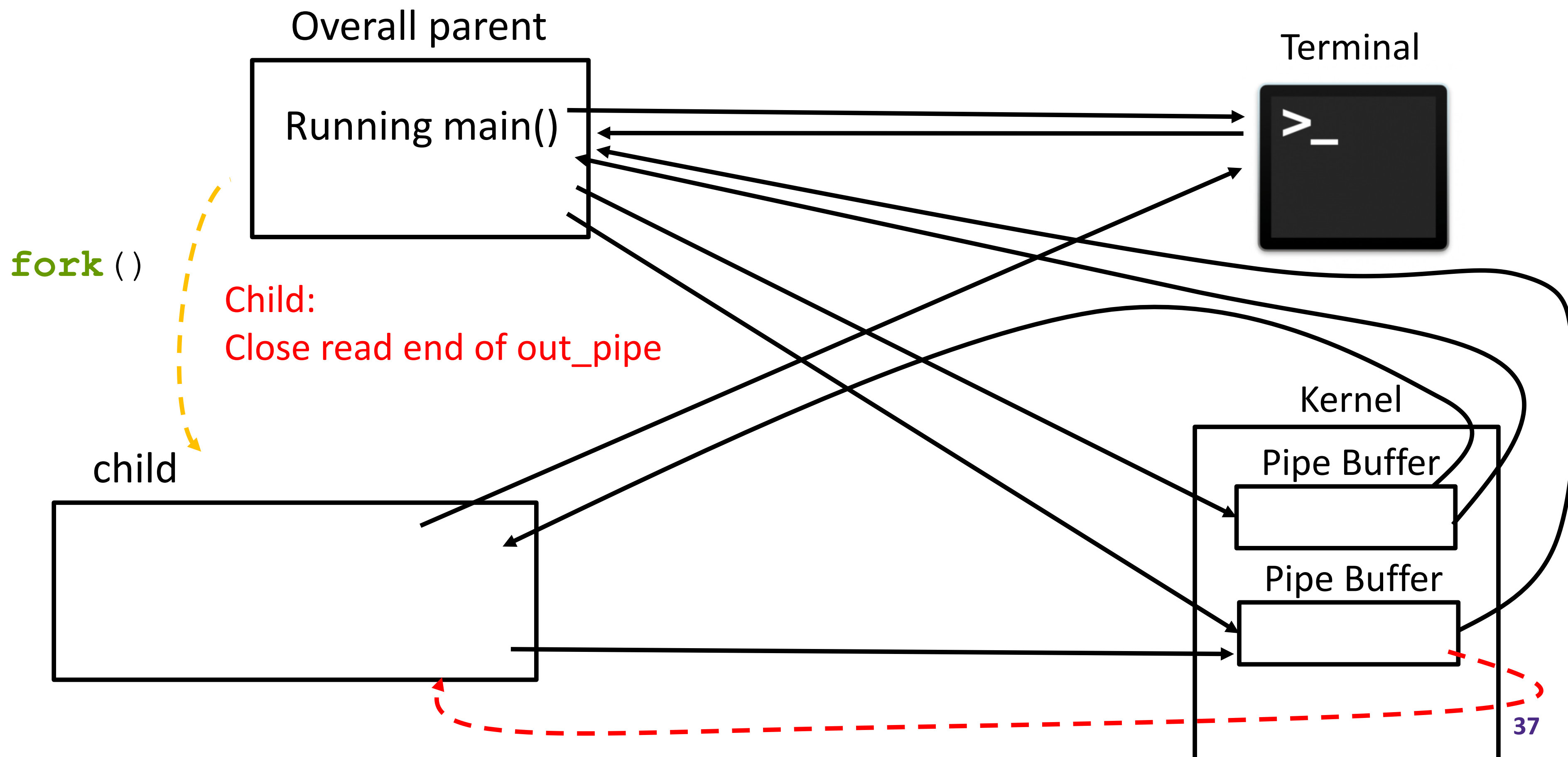
io_autograder.c Trace

- Compilation done! Run the compiled program… BUT send autograder input and capture output



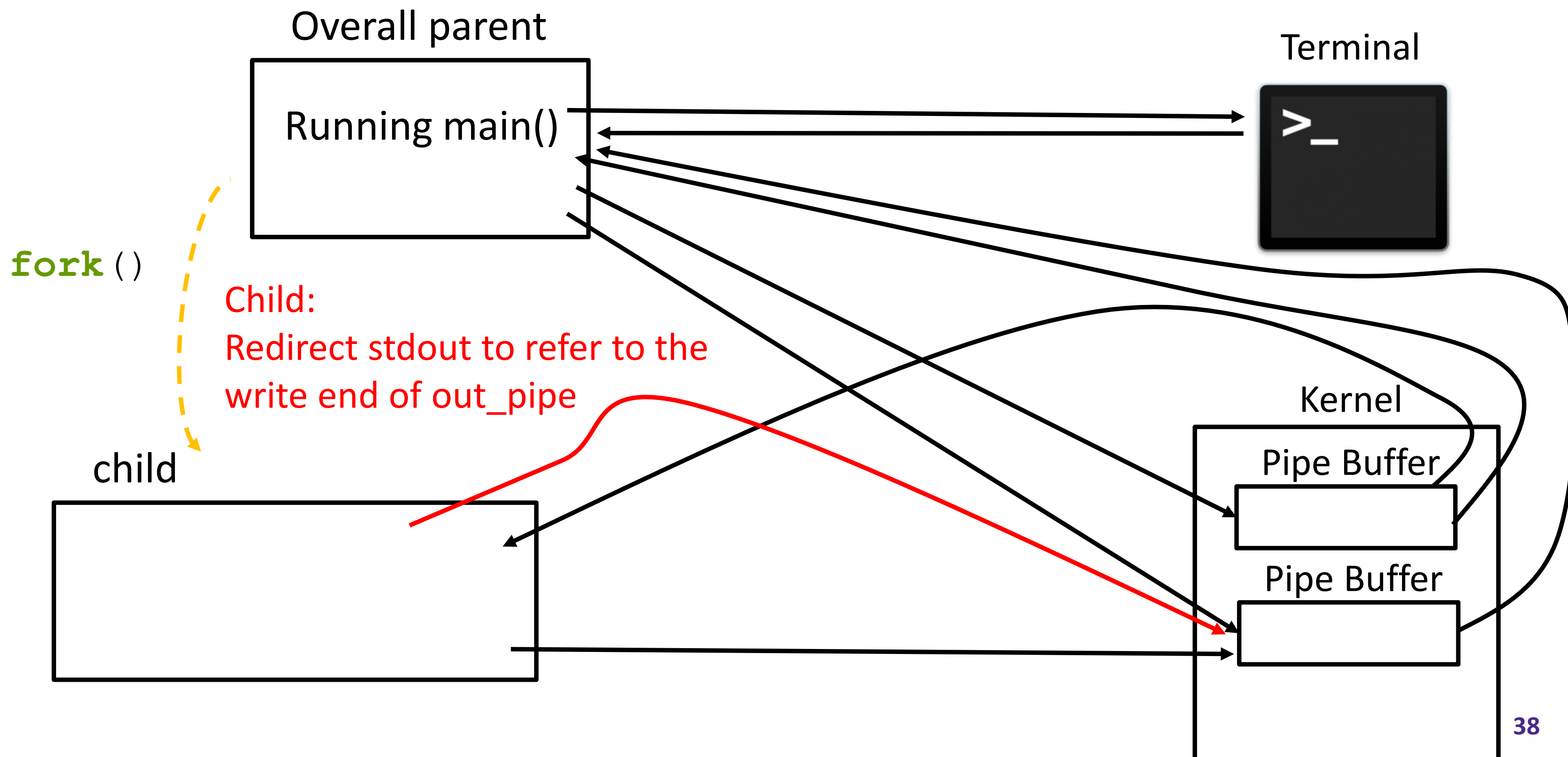
`io_autograder.c` Trace

- Compilation done! Run the compiled program… BUT send autograder input and capture output



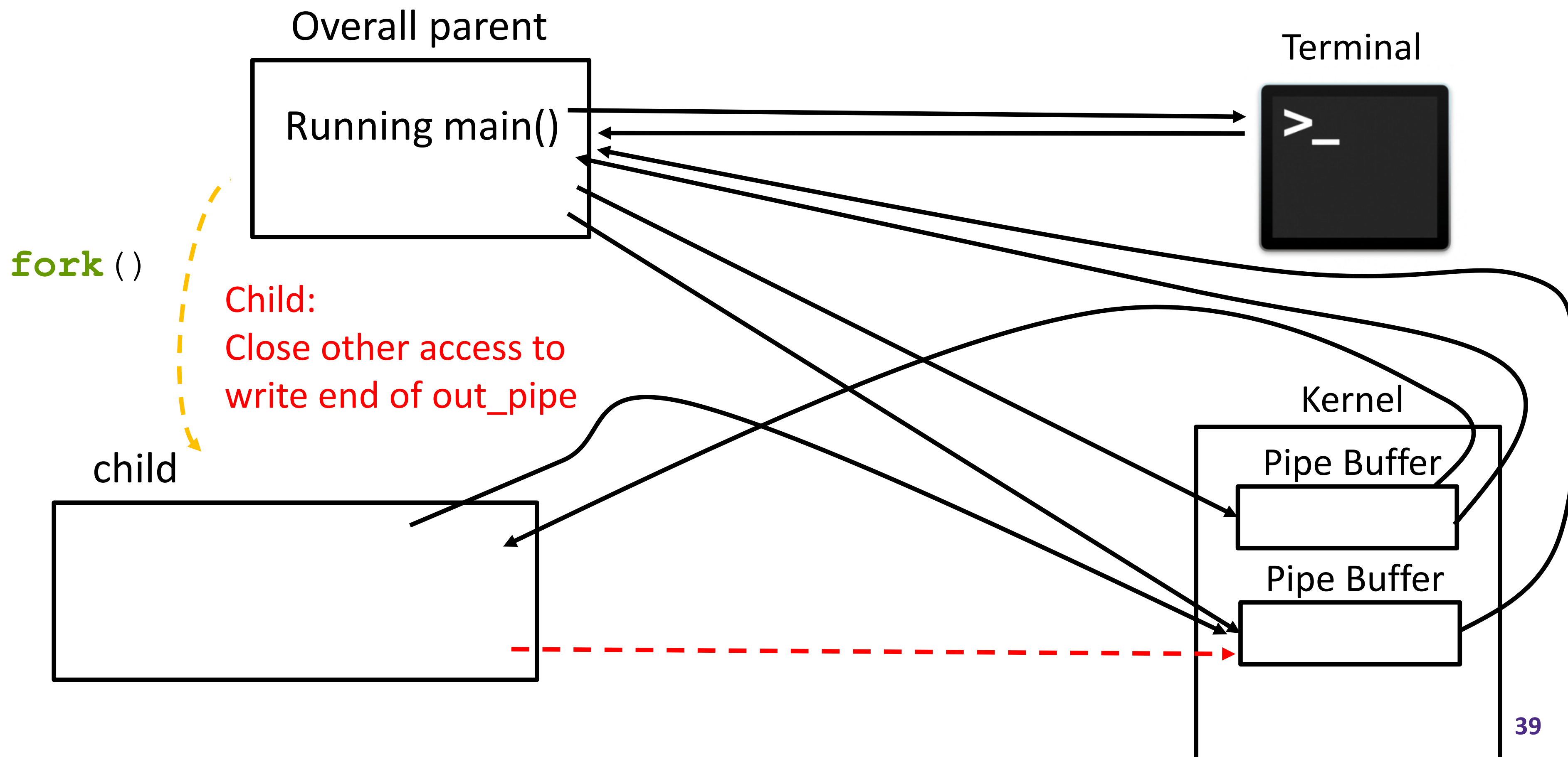
`io_autograder.c` Trace

- Compilation done! Run the compiled program… BUT send autograder input and capture output



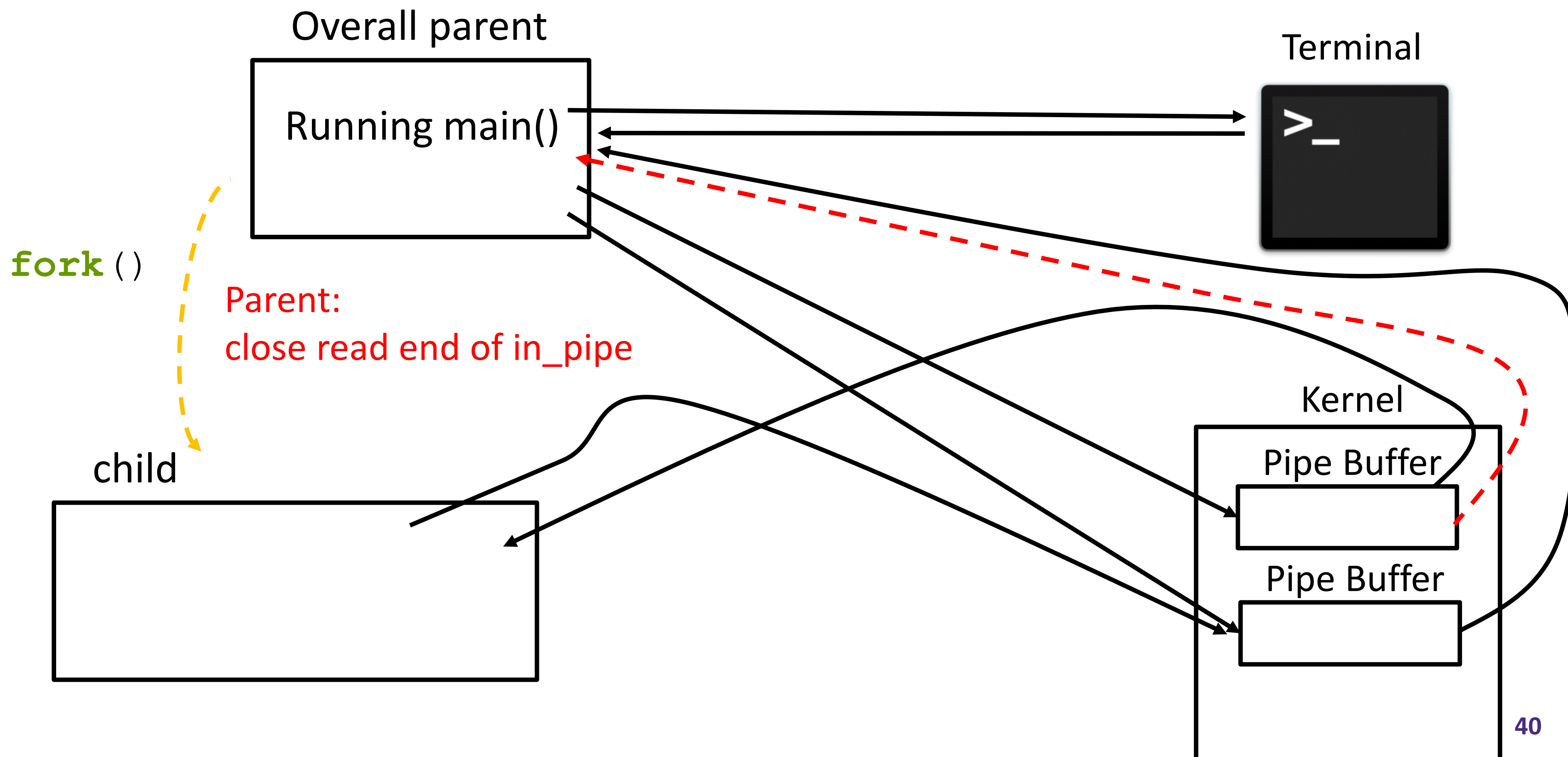
io_autograder.c Trace

- Compilation done! Run the compiled program… BUT send autograder input and capture output



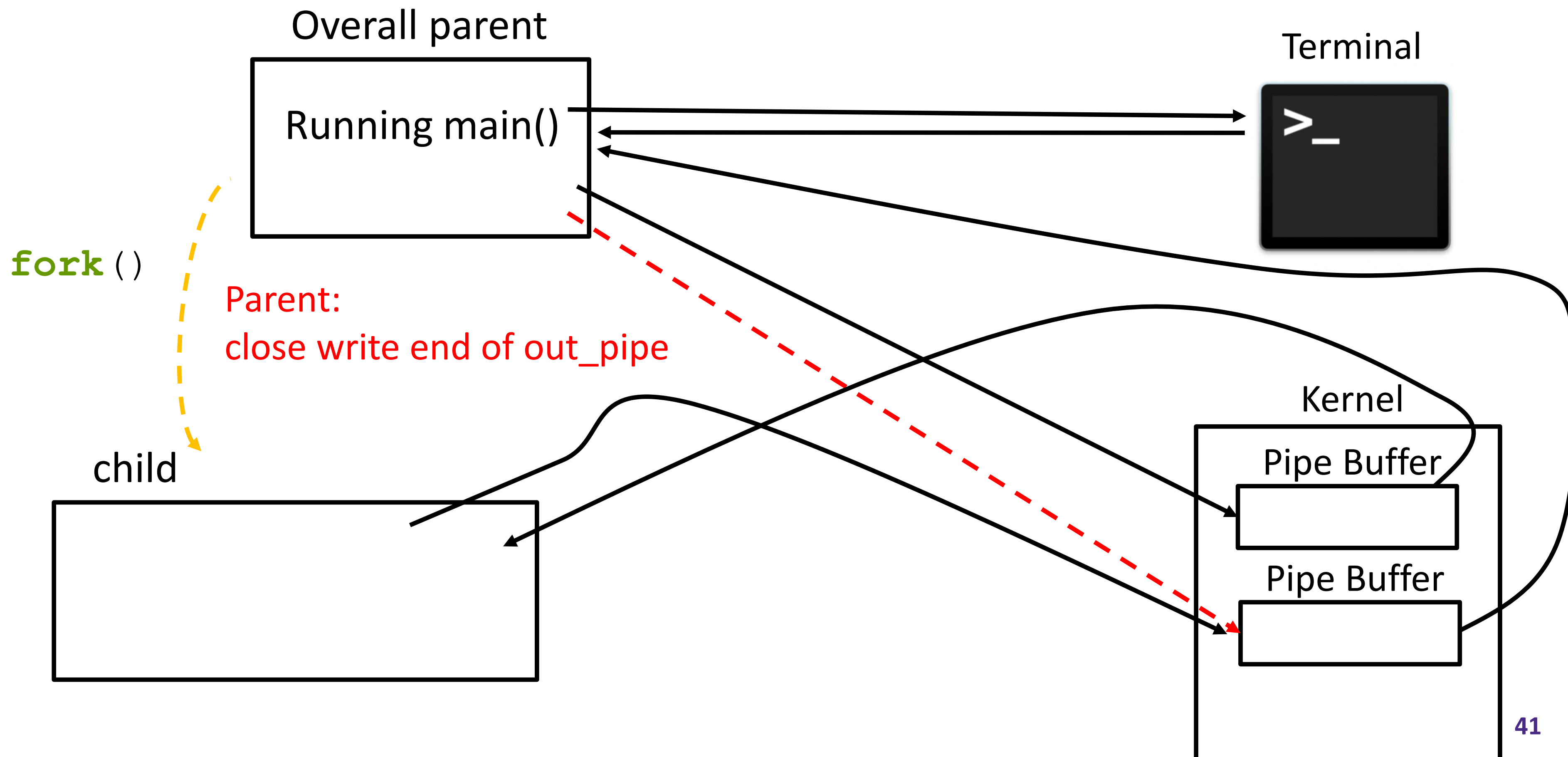
io_autograder.c Trace

- Compilation done! Run the compiled program… BUT send autograder input and capture output



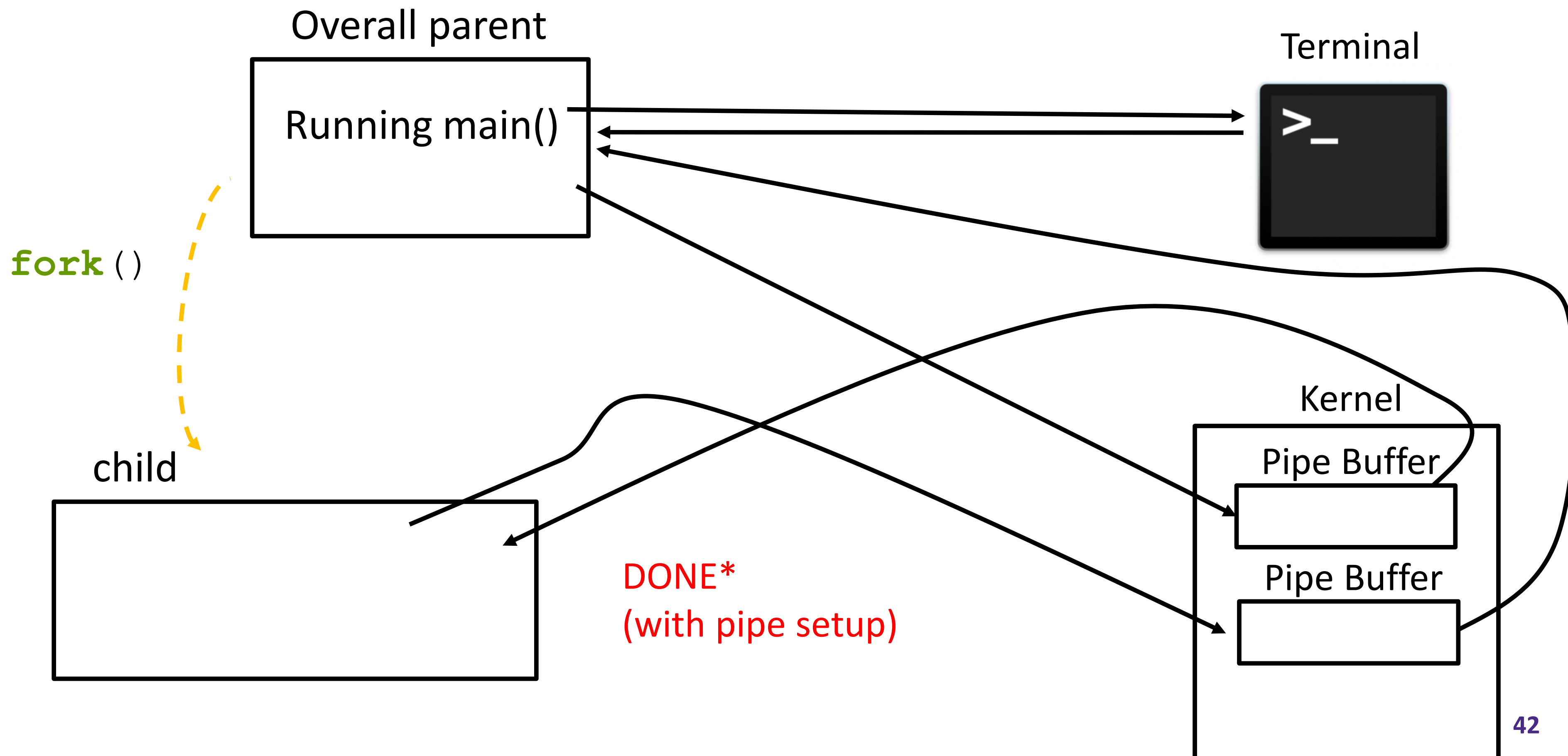
io_autograder.c Trace

- Compilation done! Run the compiled program... BUT send autograder input and capture output



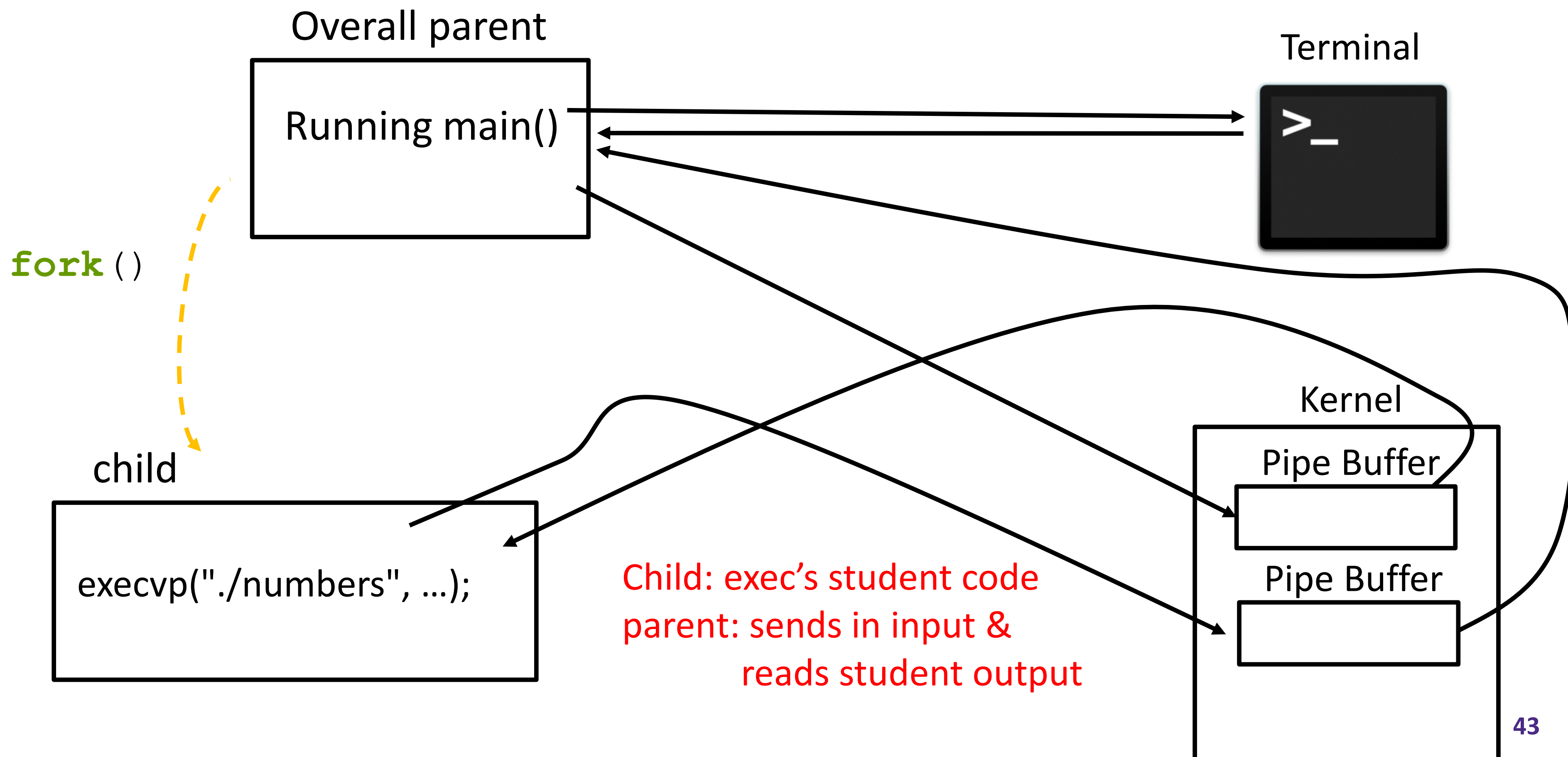
io_autograder.c Trace

- Compilation done! Run the compiled program… BUT send autograder input and capture output



io_autograder.c Trace

- Compilation done! Run the compiled program… BUT send autograder input and capture output



Lecture Outline

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

Unix Shell

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

. / ..

- "/" is used to connect directory and file names together to create a file path.
  - E.g. "**`workspace/3800/hello/`**"
- "." is used to specify the current directory.
  - E.g. "**`./test_suite`**" tells to look in the current directory for a file called "**`test_suite`**"
- ".." is like "." but refers to the parent directory.
  - E.g. "**`./solution_binaries/../test_suite`**" would be effectively the same as the previous example.

Common Commands (Pt. 1)

- "**`ls`**" lists out the entries in the specified directory (or current directory if another directory is not specified
- "**`cd`**" changes directory to the specified directory
  - E.g. "**`cd ./solution_binaries`**"
- "**`exit`**" closes the terminal
- "**`mkdir`**" creates a directory of specified name
- "**`touch`**" creates a specified file. If the file already exists, it just updates the file's time stamp

Common Commands (Pt. 2)

- "**`echo`**" takes in command line args and simply prints those args to stdout
  - "**`echo hello!`**" simply prints "**`hello!`**"
- "**`wc`**" reads a file or from stdin some contents. Prints out the line count, word count, and byte count
- "**`cat`**" prints out the contents of a specified file to stdout. If no file is specified, prints out what is read from stdin
- "**`head`**" print the first 10 line of specified file or stdin to stdout

Common Commands (Pt. 3)

- "**`grep`**" given a pattern (regular expression) searches for all occurrences of such a pattern. Can search a file, search a directory recursively or stdin. Results printed to stdout

- "**`history`**" prints out the history of commands used by you on the terminal

- "**`cron`**" a program that regularly checks for and runs any commands that are scheduled via "crontab"

- "**`wget`**" specify a URL, and it will download that file for you

Unix Shell Commands

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

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

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

Unix Shell Control Operators

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

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

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

Unix Shell Control Operators

- `cmd < file`, redirects stdin to instead read from the specified file
  - E.g. "`./penn-shredder < test_case`"

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

- Complex example:

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

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.

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…

Correctly gets the number of files, but not ONLY the number of files

`ls | wc -l`
would be preferred.

Penn-Shell (Proj1) Overview

- In penn-shell milestone, 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
  - We will provide the parser for you

- Demo in class on Tuesday next week (9/19)

Unix Shell Control Operators: Pipe

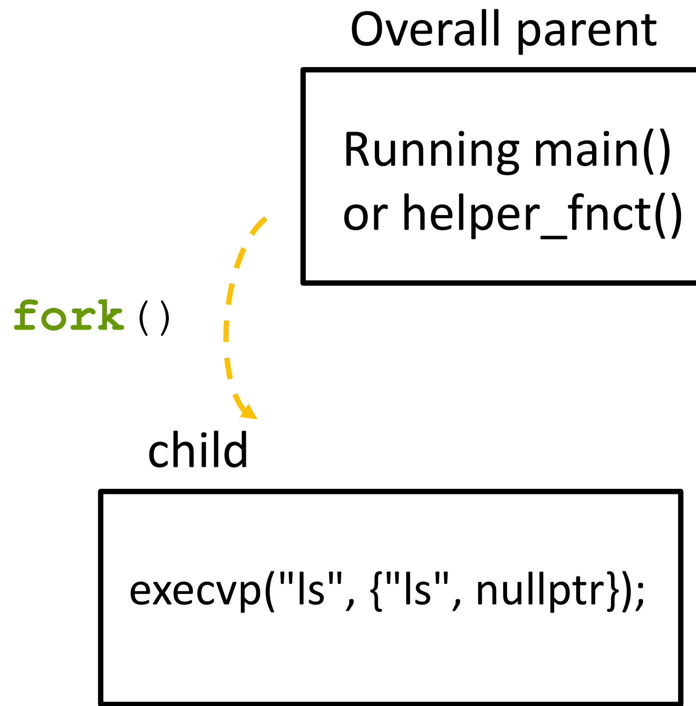
- `cmd1 | cmd2`, creates a pipe so that the stdout of cmd1 is redirected to the stdin of cmd2
  - E.g. "`history | grep valgrind`"

Suggested Approach

- HIGHLY ENCOURAGED to follow this suggested approach
  - Write a program that "echo" stdin
  - Write a program that can handle commands with no pipes
    - `"ls"`
  - Add support for 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"`

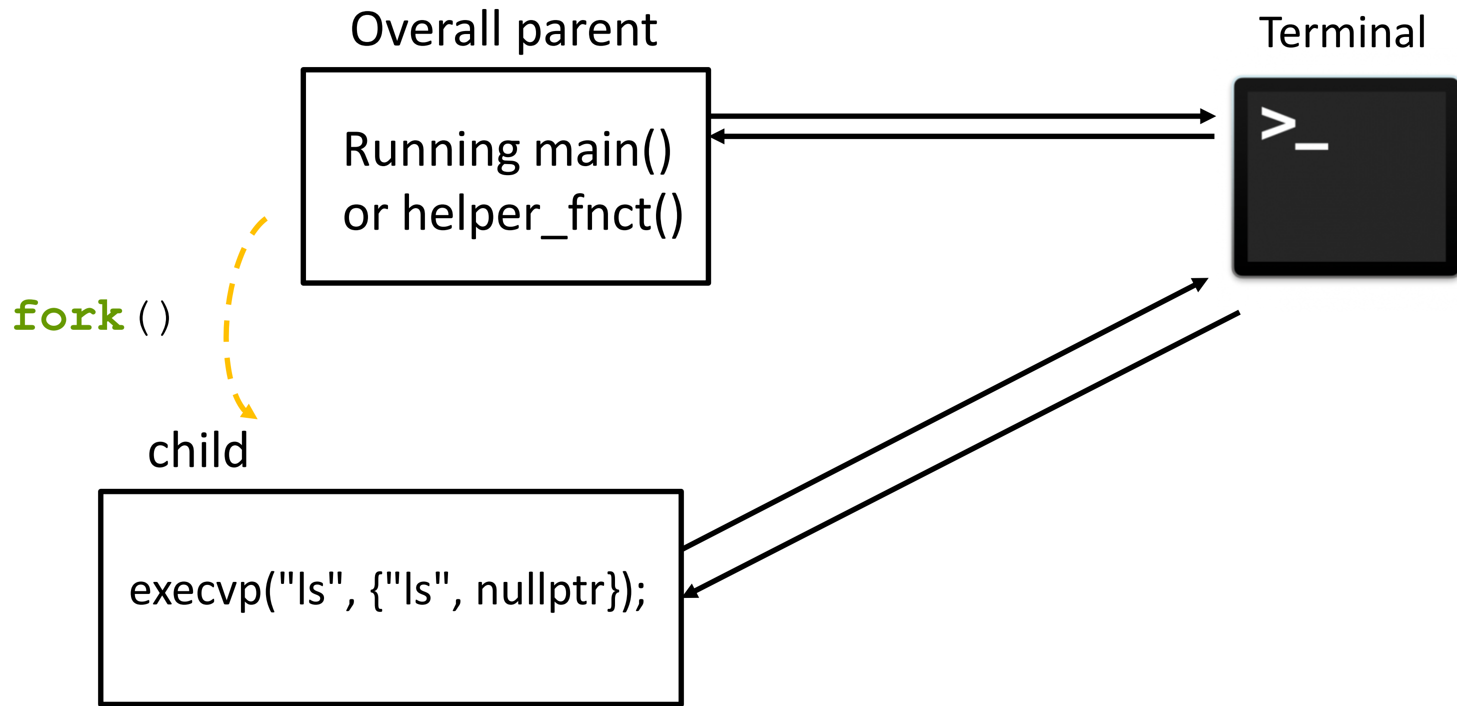
`penn-shell` Example Line

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



`penn-shell` Example Line

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

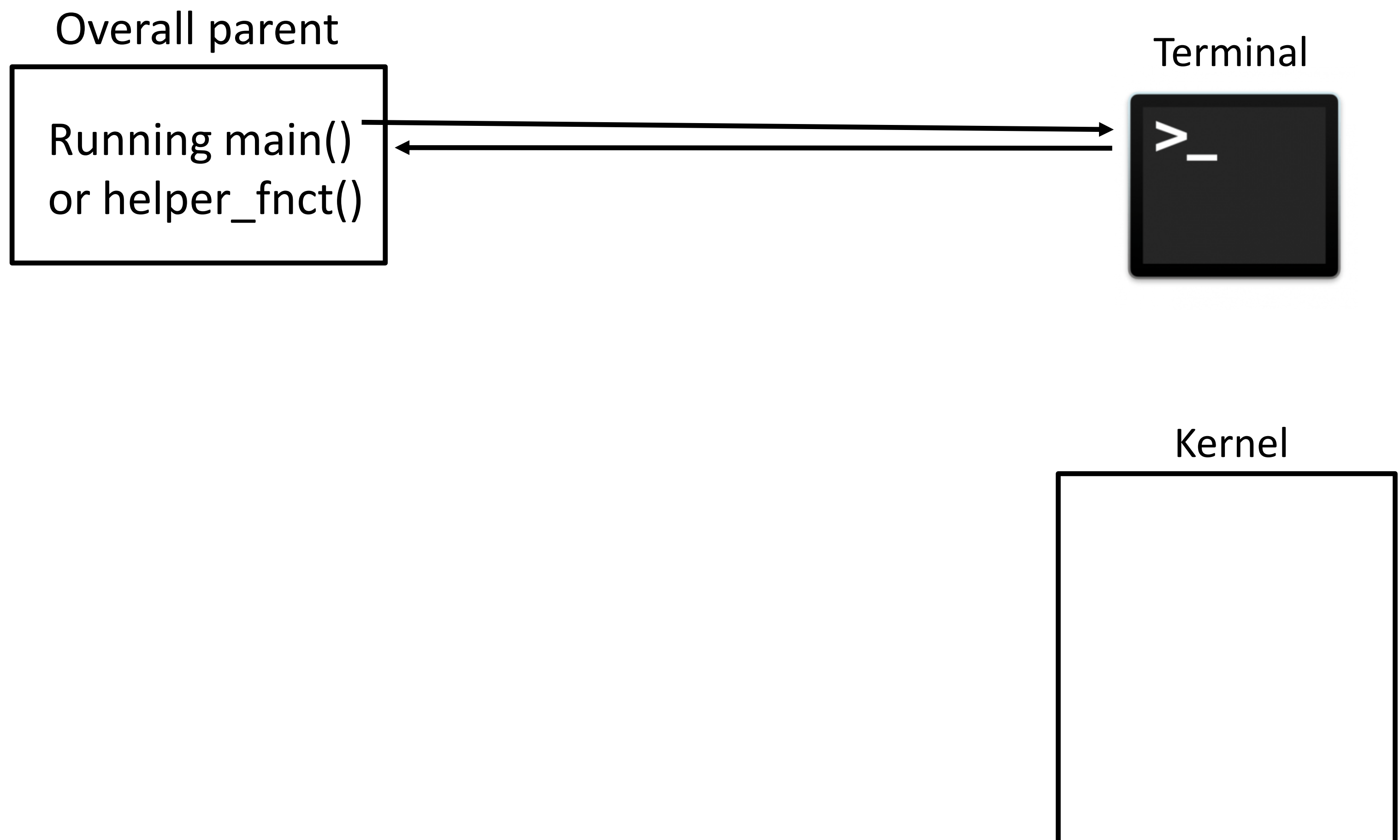


`penn-shell` Hints

- 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

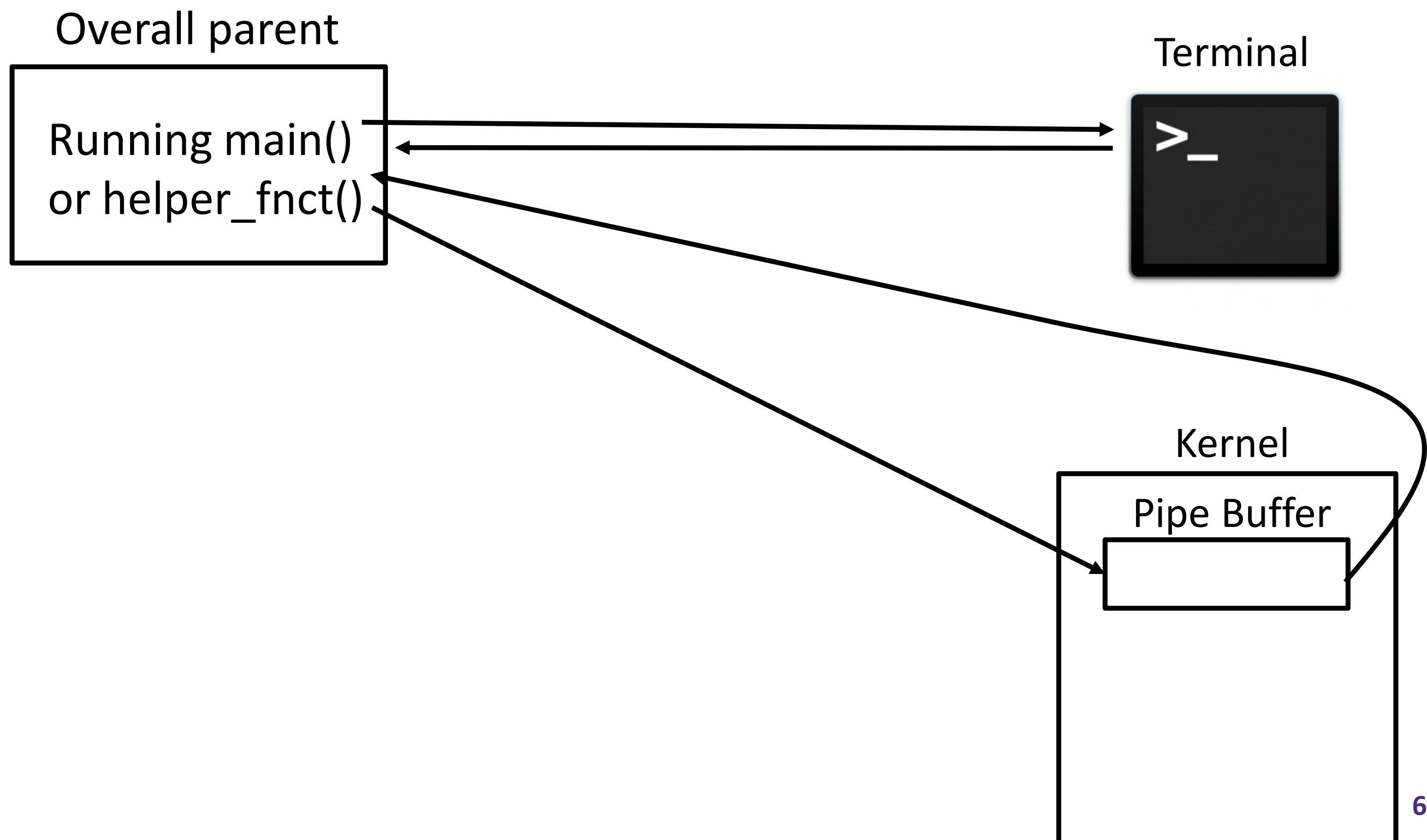
`penn-shell` Example Line 2

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



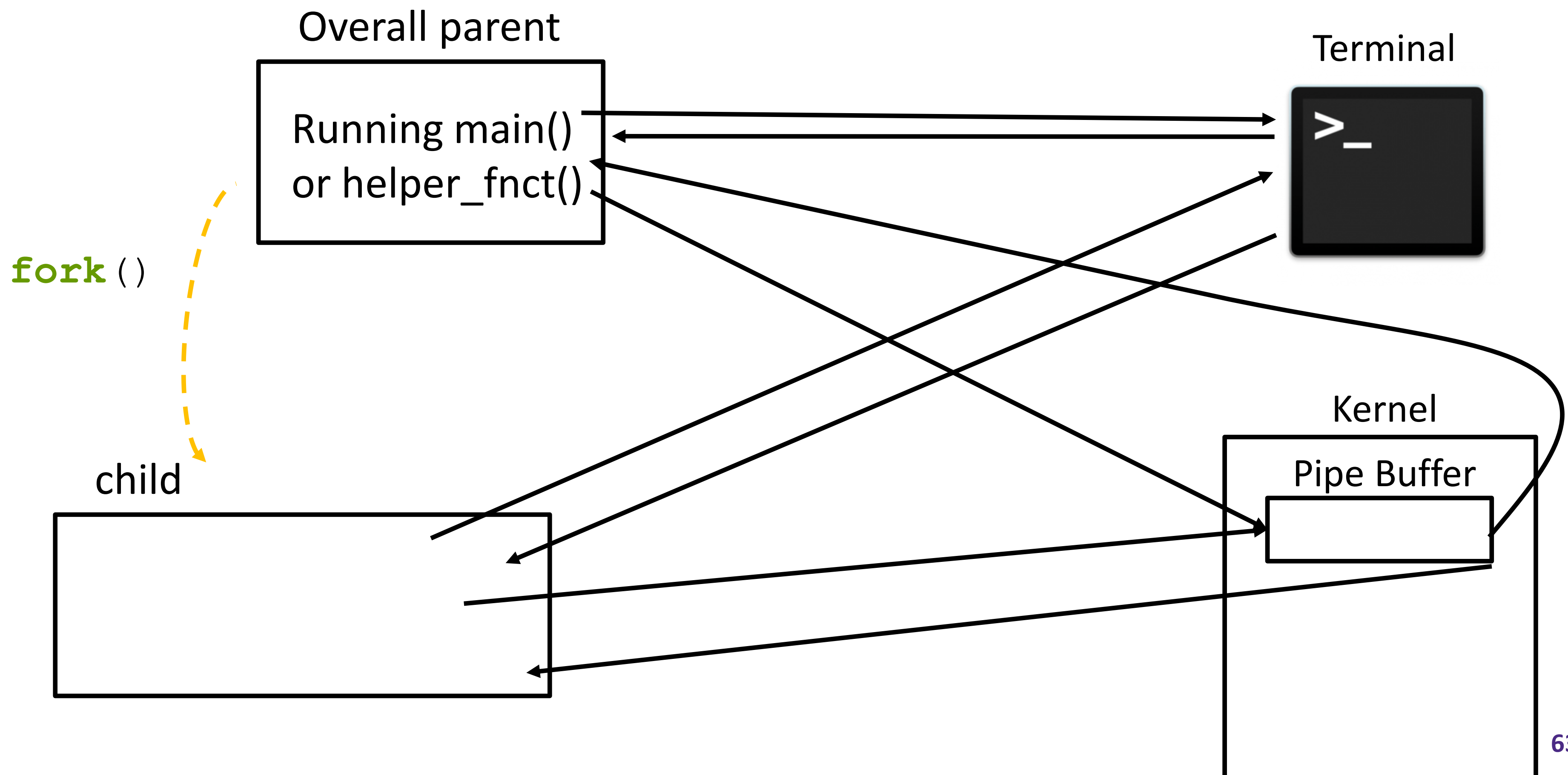
`penn-shell` Example Line 2

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



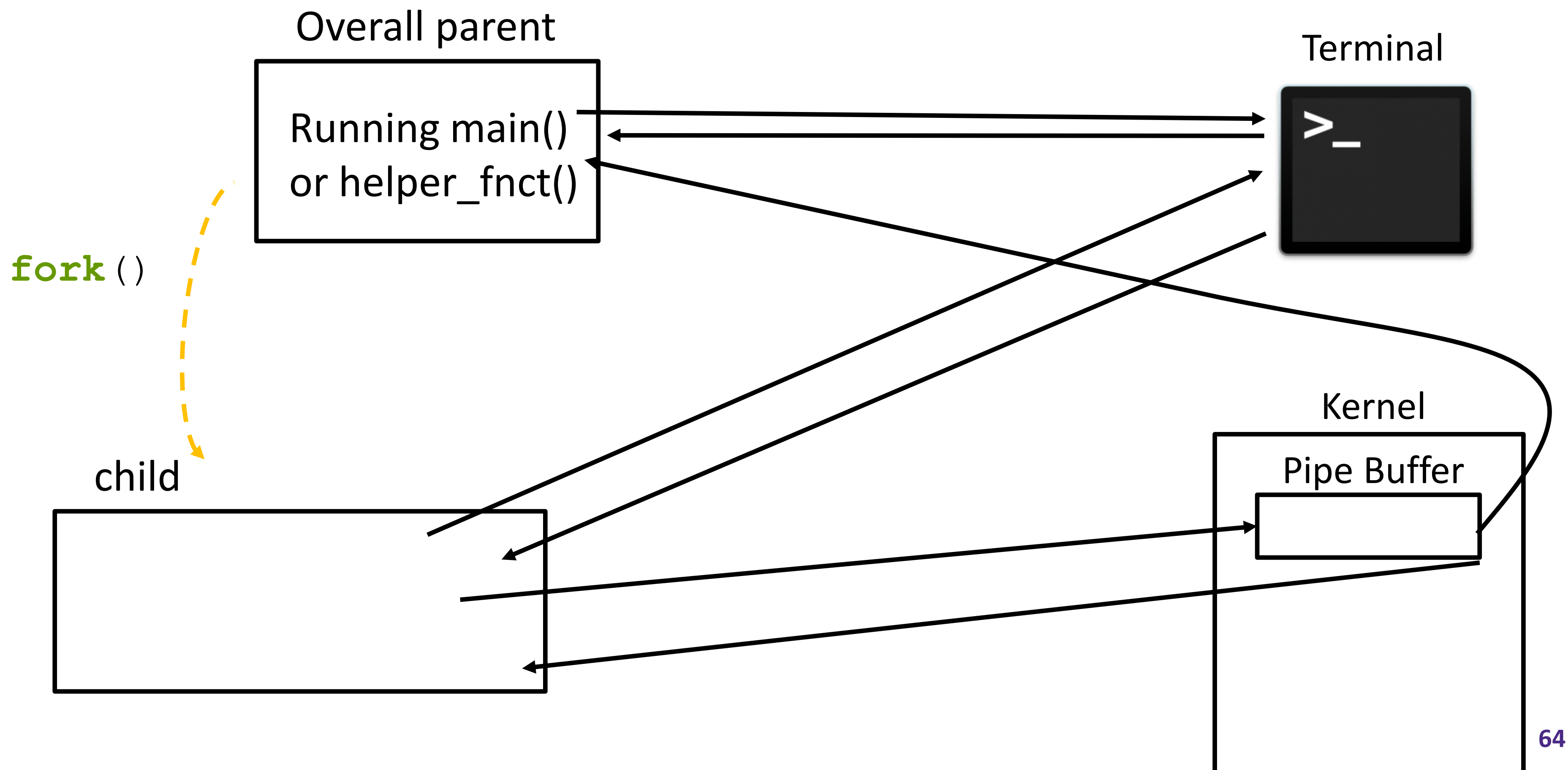
penn-shell Example Line 2

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



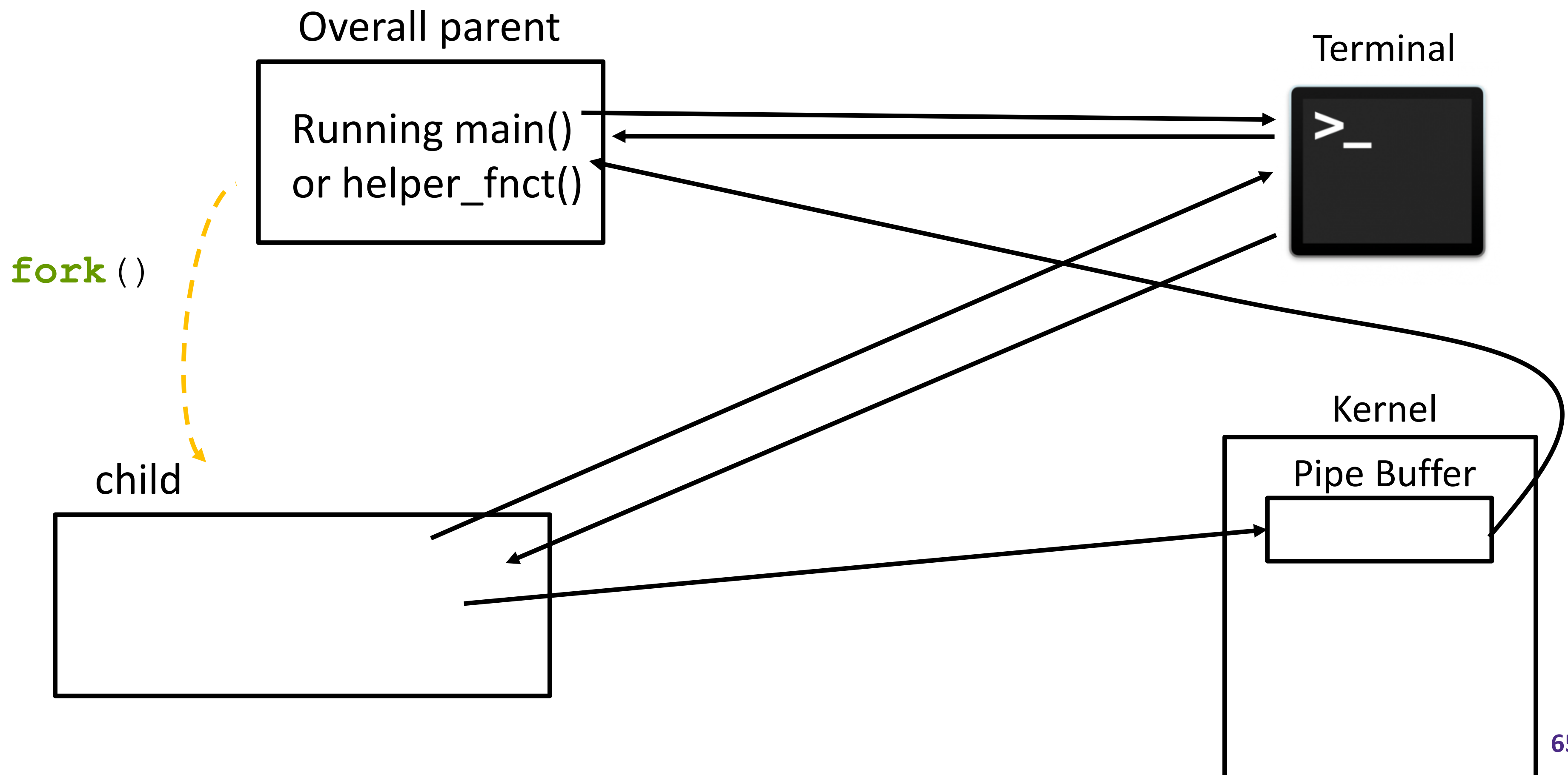
`penn-shell` Example Line 2

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



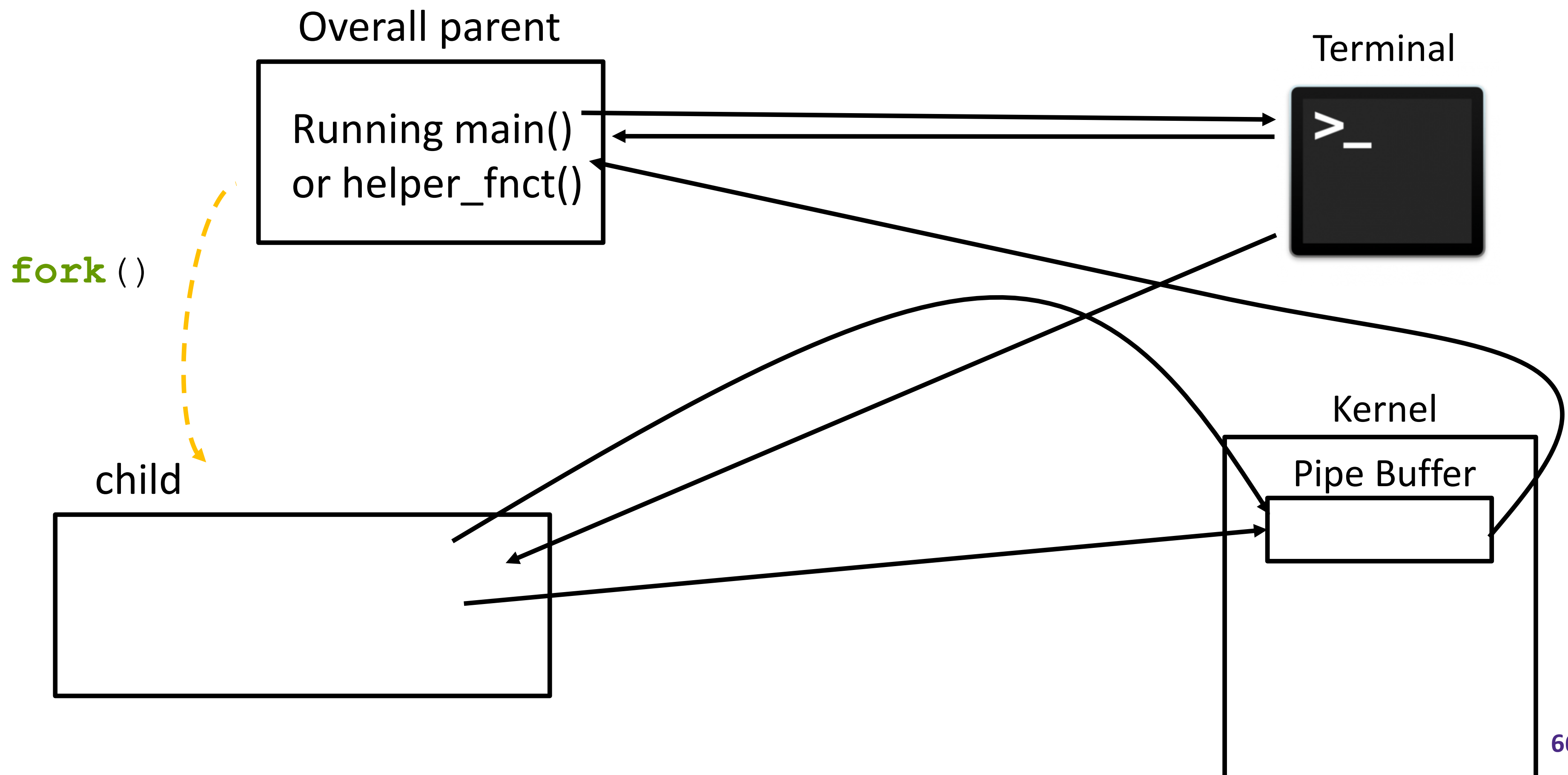
`penn-shell` Example Line 2

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



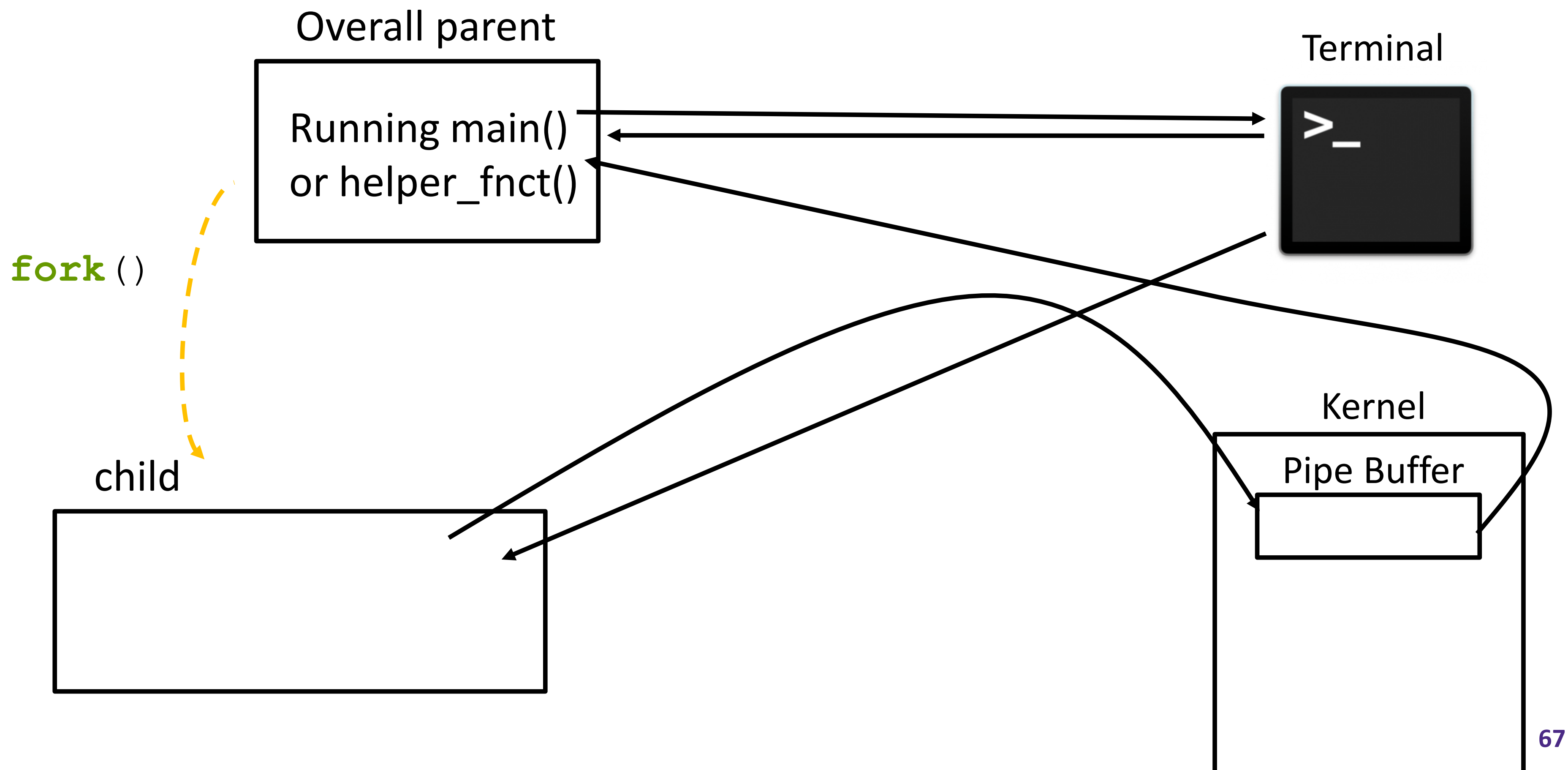
`penn-shell` Example Line 2

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



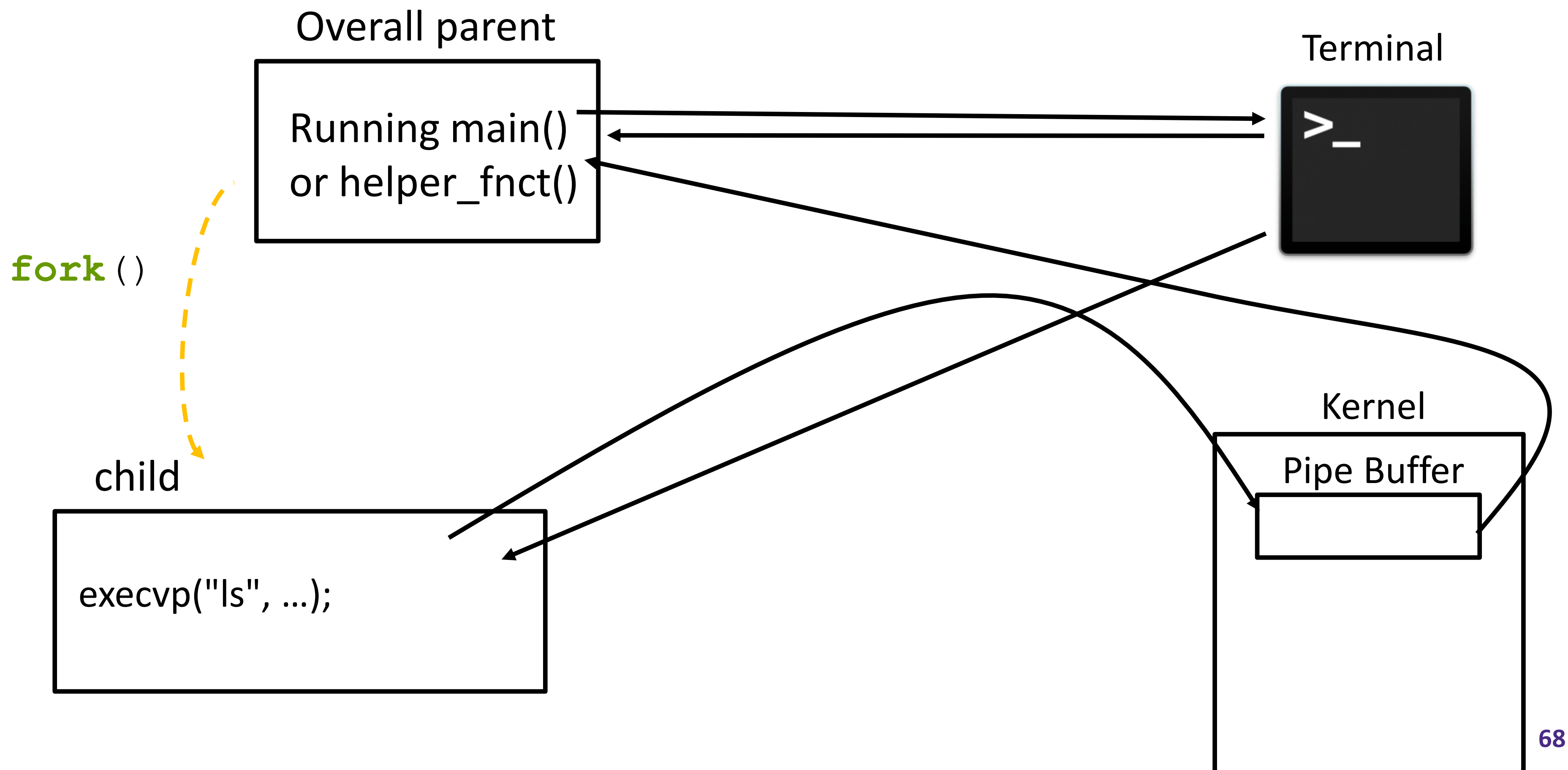
penn-shell Example Line 2

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



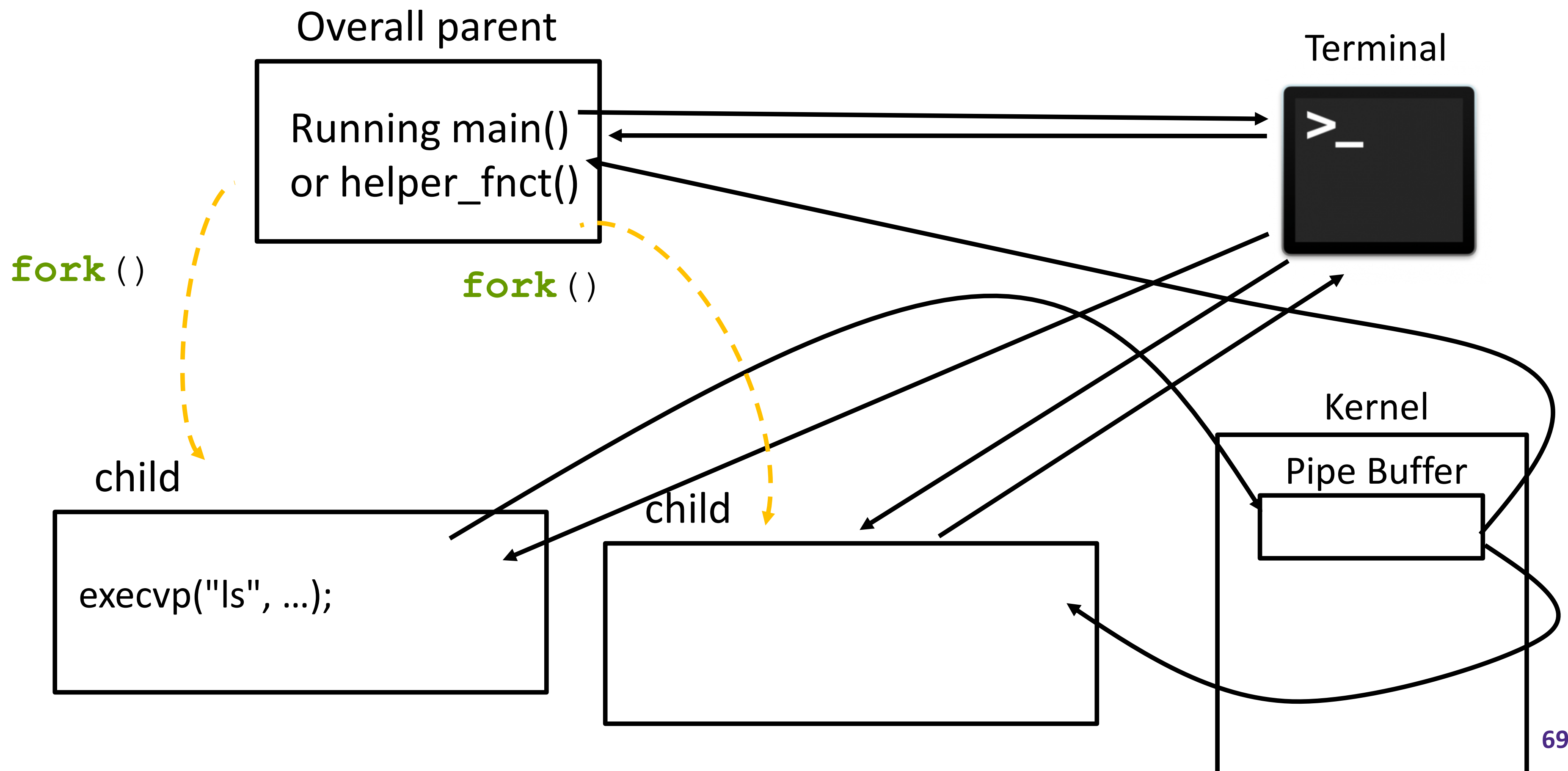
penn-shell Example Line 2

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



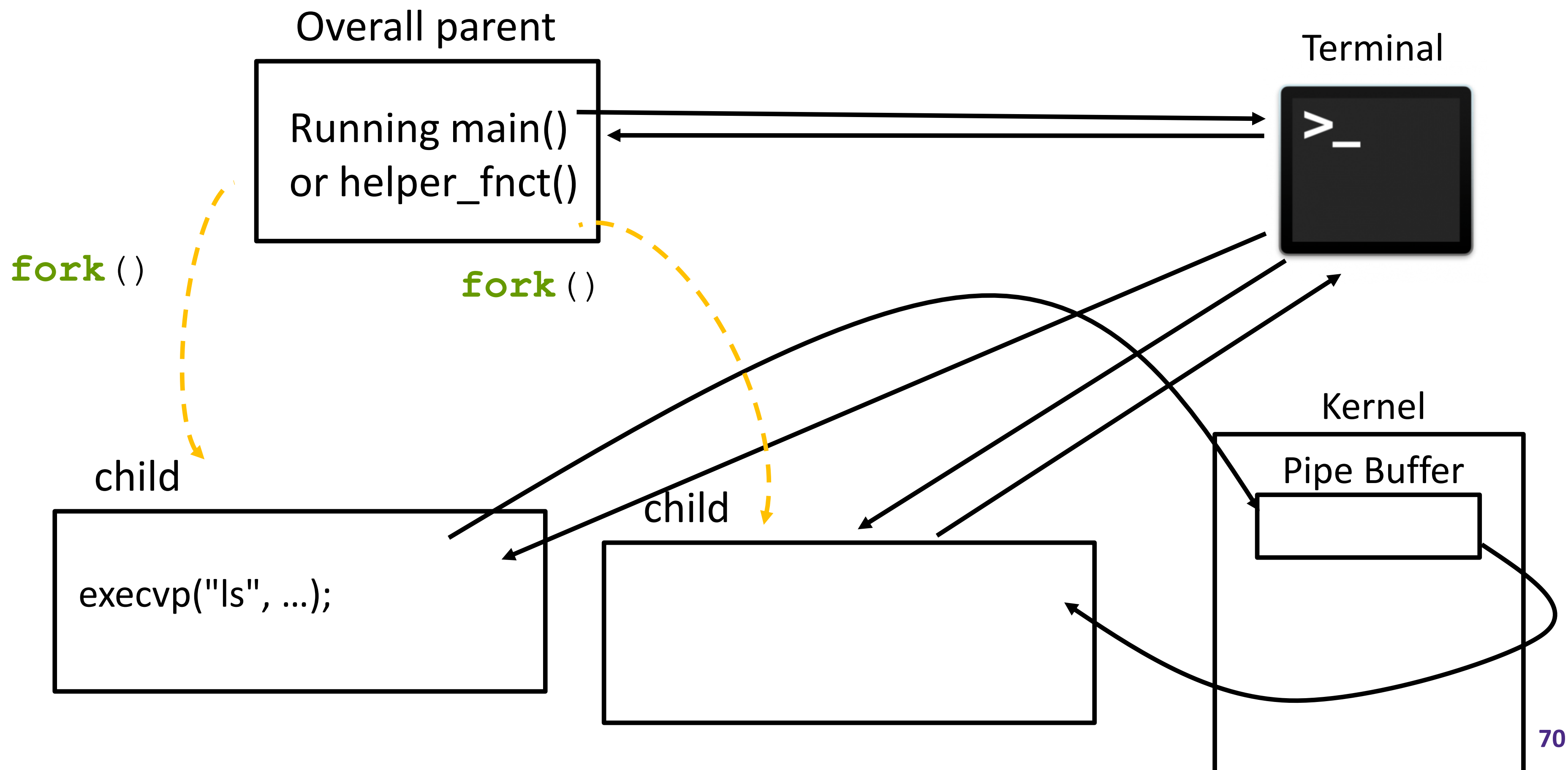
`penn-shell` Example Line 2

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



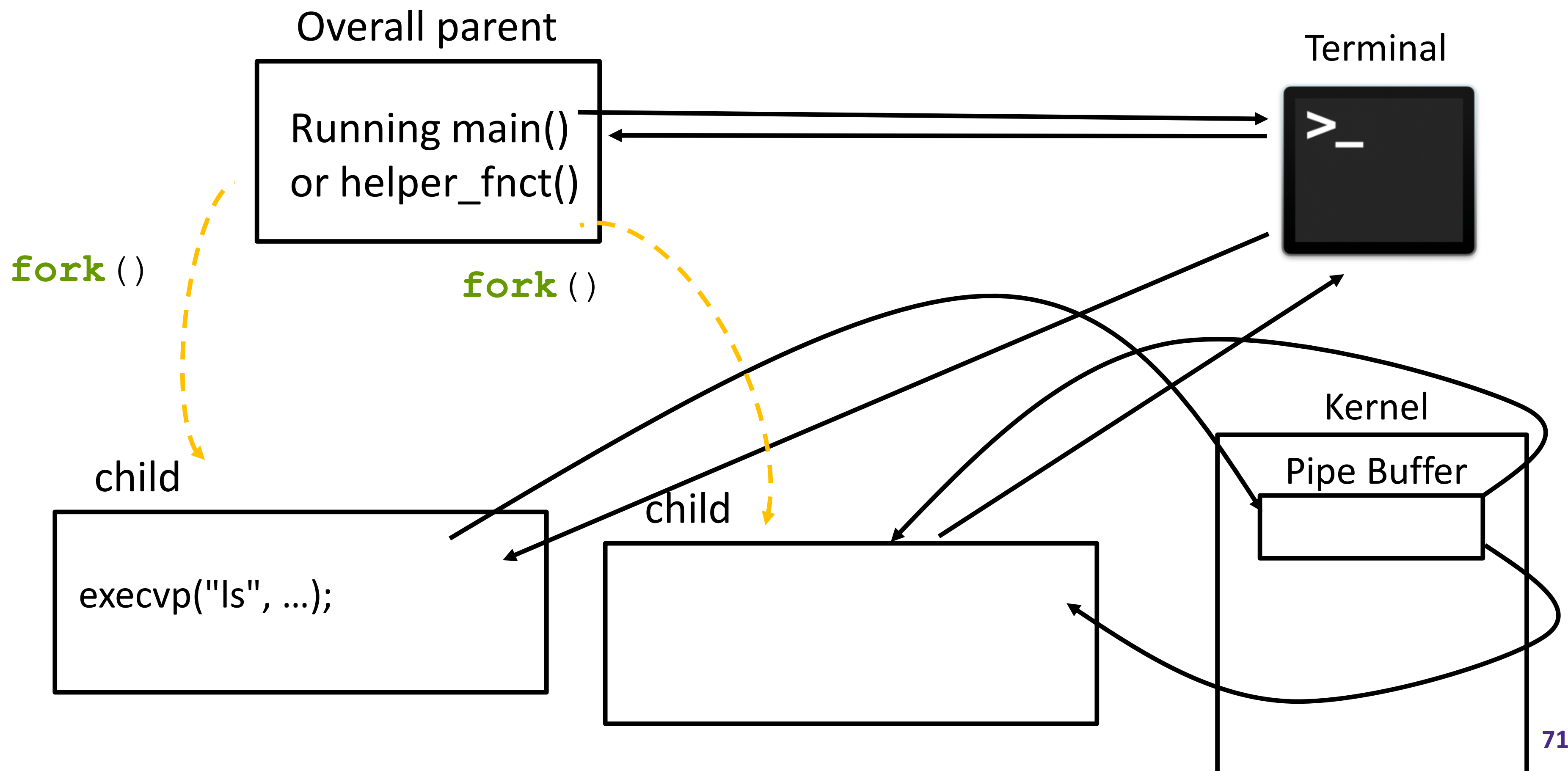
`penn-shell` Example Line 2

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



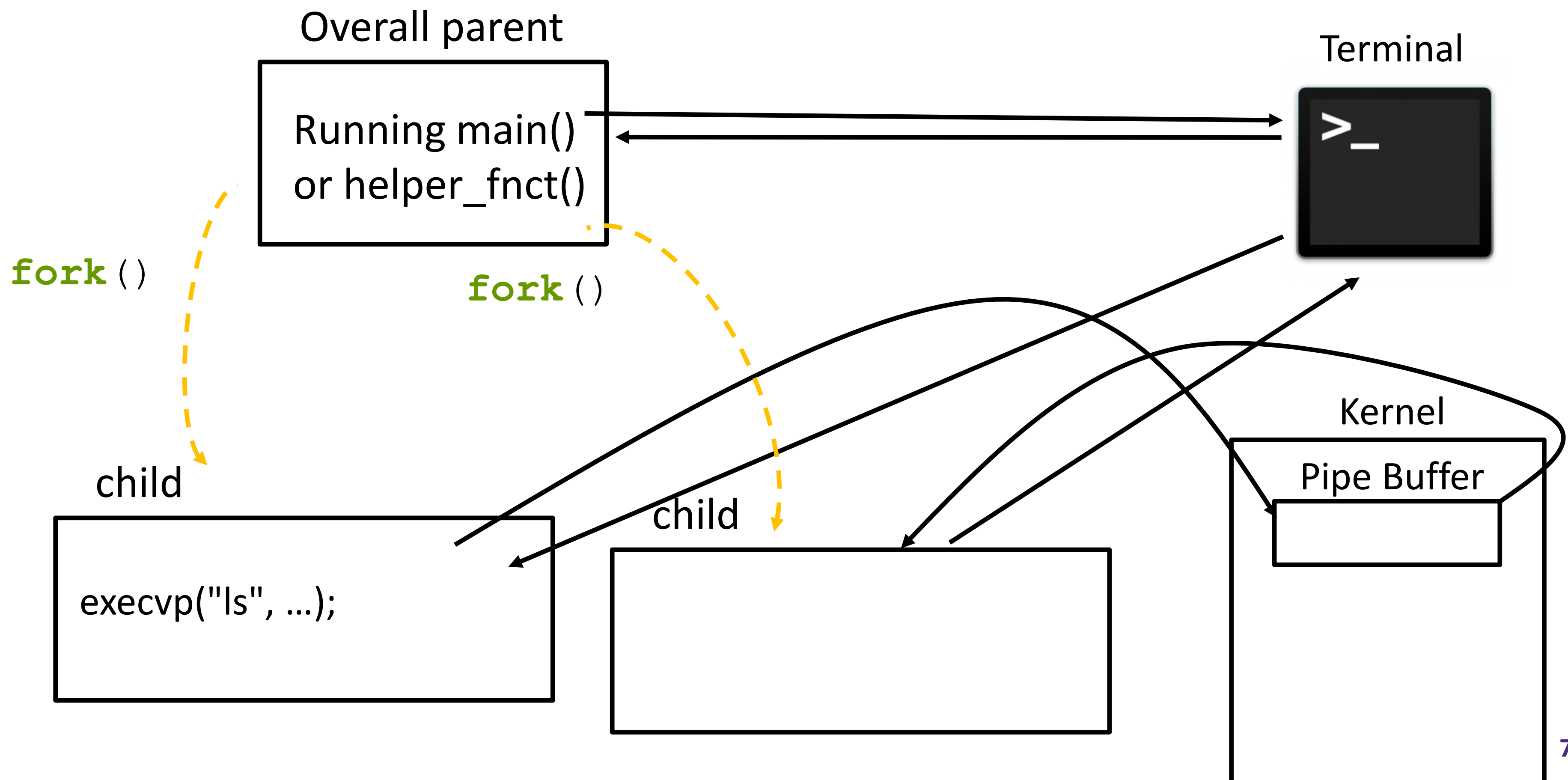
penn-shell Example Line 2

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



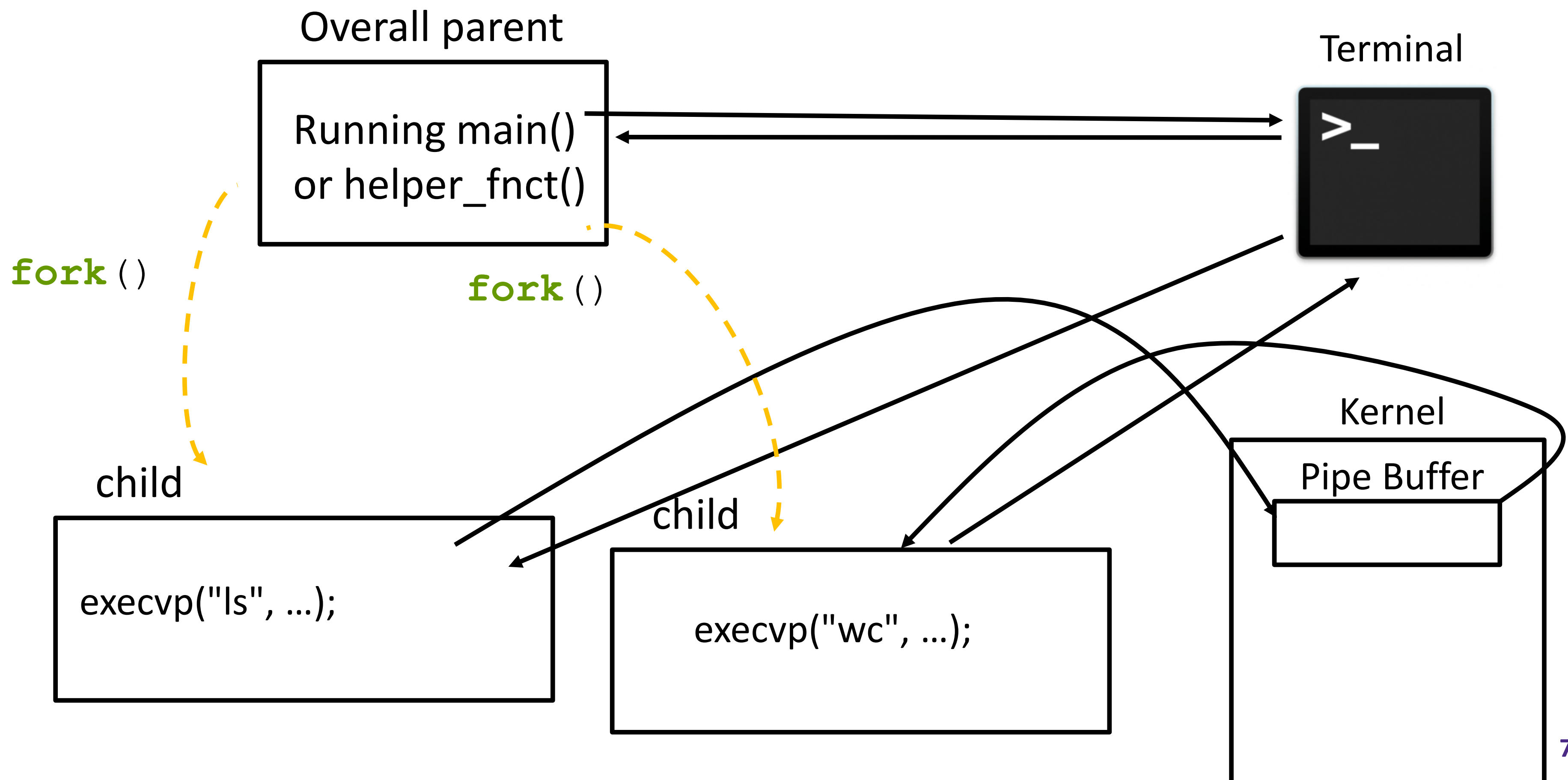
`penn-shell` Example Line 2

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



`penn-shell` Example Line 2

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

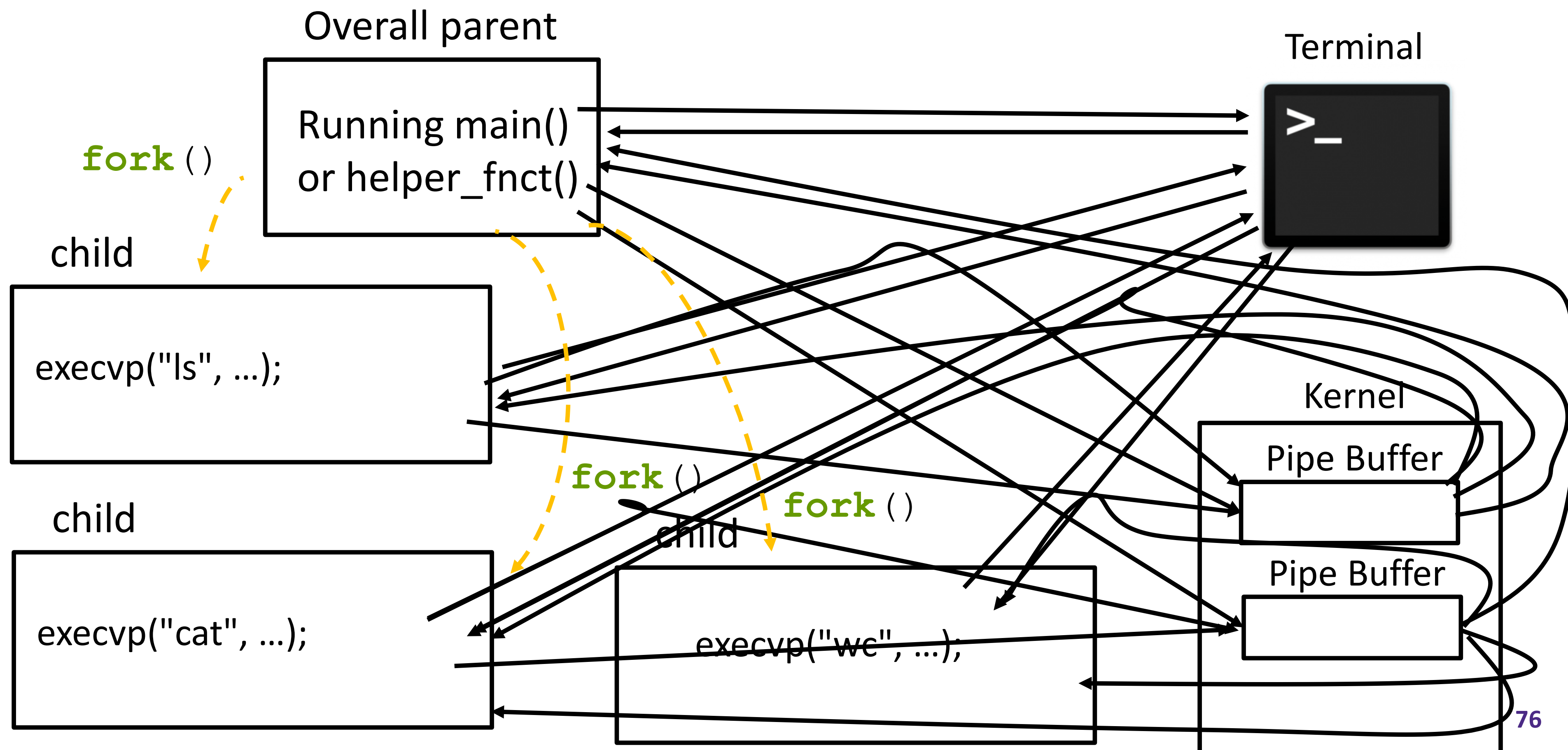


pollev.com/tqm

- What does the code `two_pipes.c` from the website do?
  - I promise it works, no deadlock or crashing etc.

`penn-shell` Example Line 3

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



`penn-shell` Hints pt. 2

- Pipes can all be created at the start or only as you need them.
  - `two_pipes.c` creates pipes "as you need them" and closes things ASAP

- Pipes can be closed as early as possible or more lazily
  - make sure that all ends that aren't needed by a process are closed before it potentially blocks, especially the write end of the pipes

- Can do this either iteratively or recursively, whatever makes more sense to you ☺

Advice

- Don’t get discouraged, this looks hard, but it is not that bad
- Reference the example code posted along this on the website
- You have a partner to do this with
- The TAs & I are here to help
- Come back to these slides for help