Pipe() & HW4

Computer Systems Programming, Spring 2024

Instructor: Travis McGaha

TAs:

Ash Fujiyama
CV Kunjeti
Felix Sun
Heyi Liu
Kevin Bernat
Lang Qin
Sean Chuang
Serena Chen
Yuna Shao

Logistics

- HW03 due Friday this week
  - Recitation last week had an overview of what it is doing
  - Autograder is posted
  - Travis has extra oh from 2 to 3:30 on Friday

- Project code posted
  - Due May 1st @ 11:59pm
  - There is a component that is graded by hand
  - Git repositories to be created soon
  - Beginning of this lecture helps with setup.

- Next Checkin to be released soon

Poll Everywhere

pollev.com/tqm

- Any questions?

Lecture Outline

- **Pipe**
- Unix Shell
- HW4

Pipes

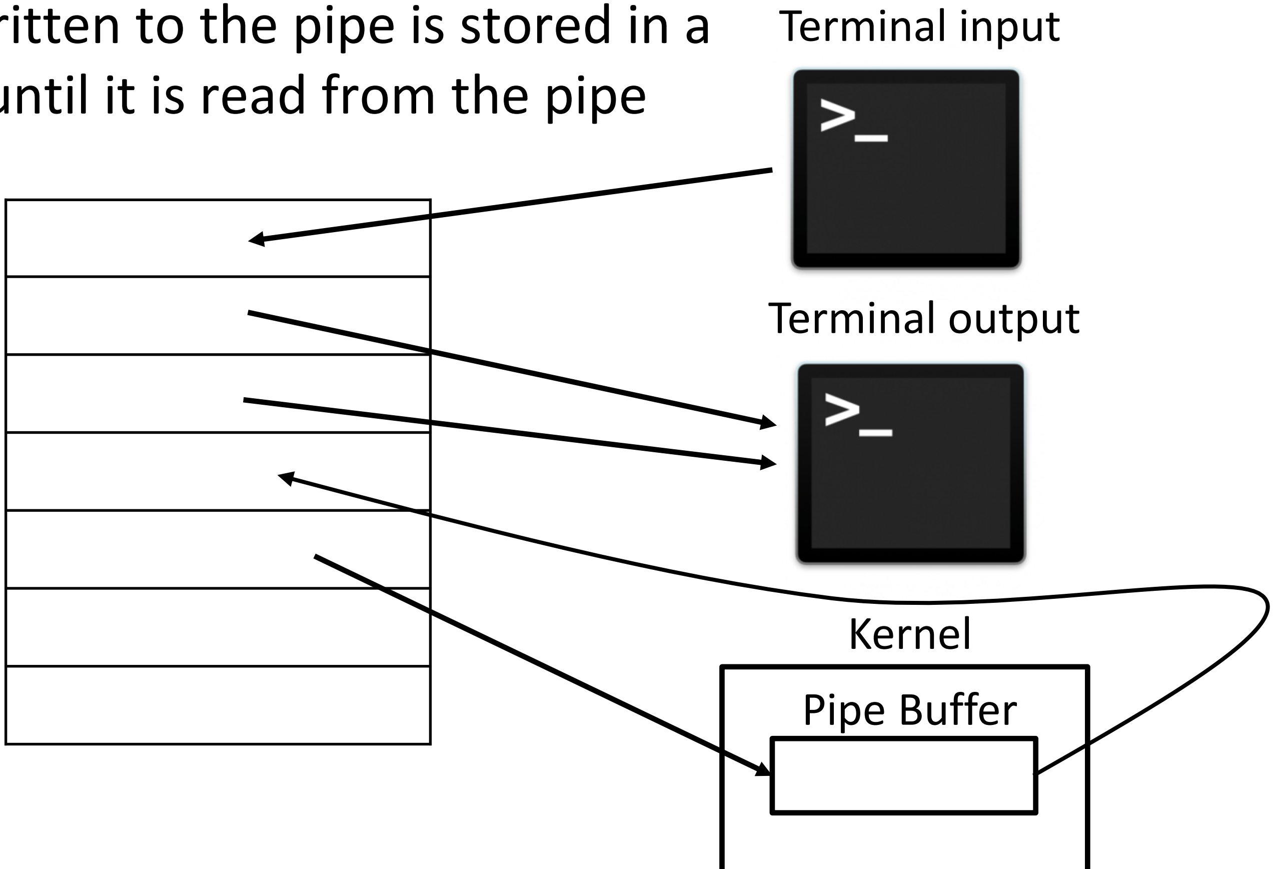
```
int pipe(int pipefd[2]);
```

- 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

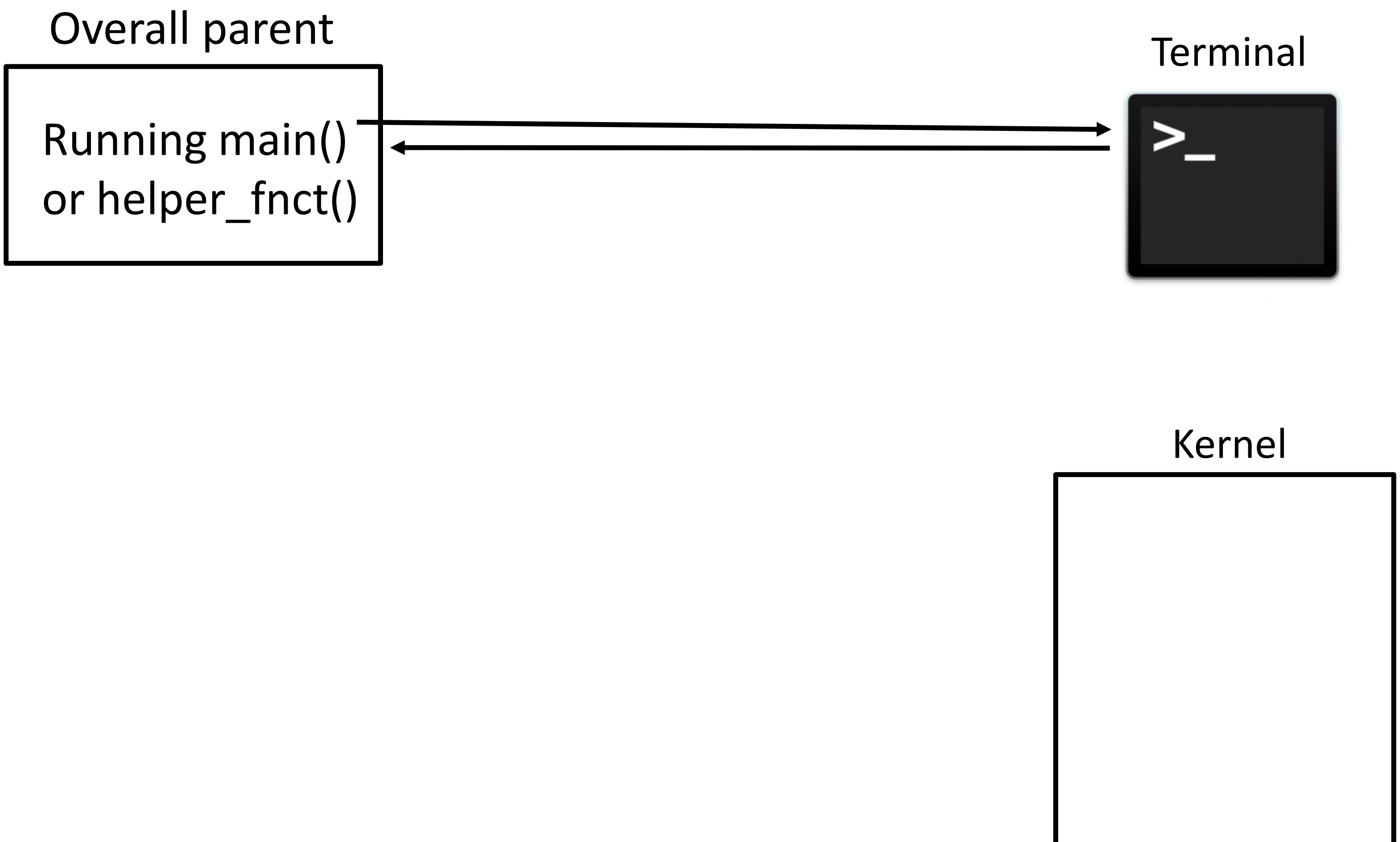
- A pipe can be thought of as a "file" that has distinct file descriptors for reading and writing. This "file" only exists as long as the pipe exists and is maintained by the OS.
  - Data written to the pipe is stored in a buffer until it is read from the pipe



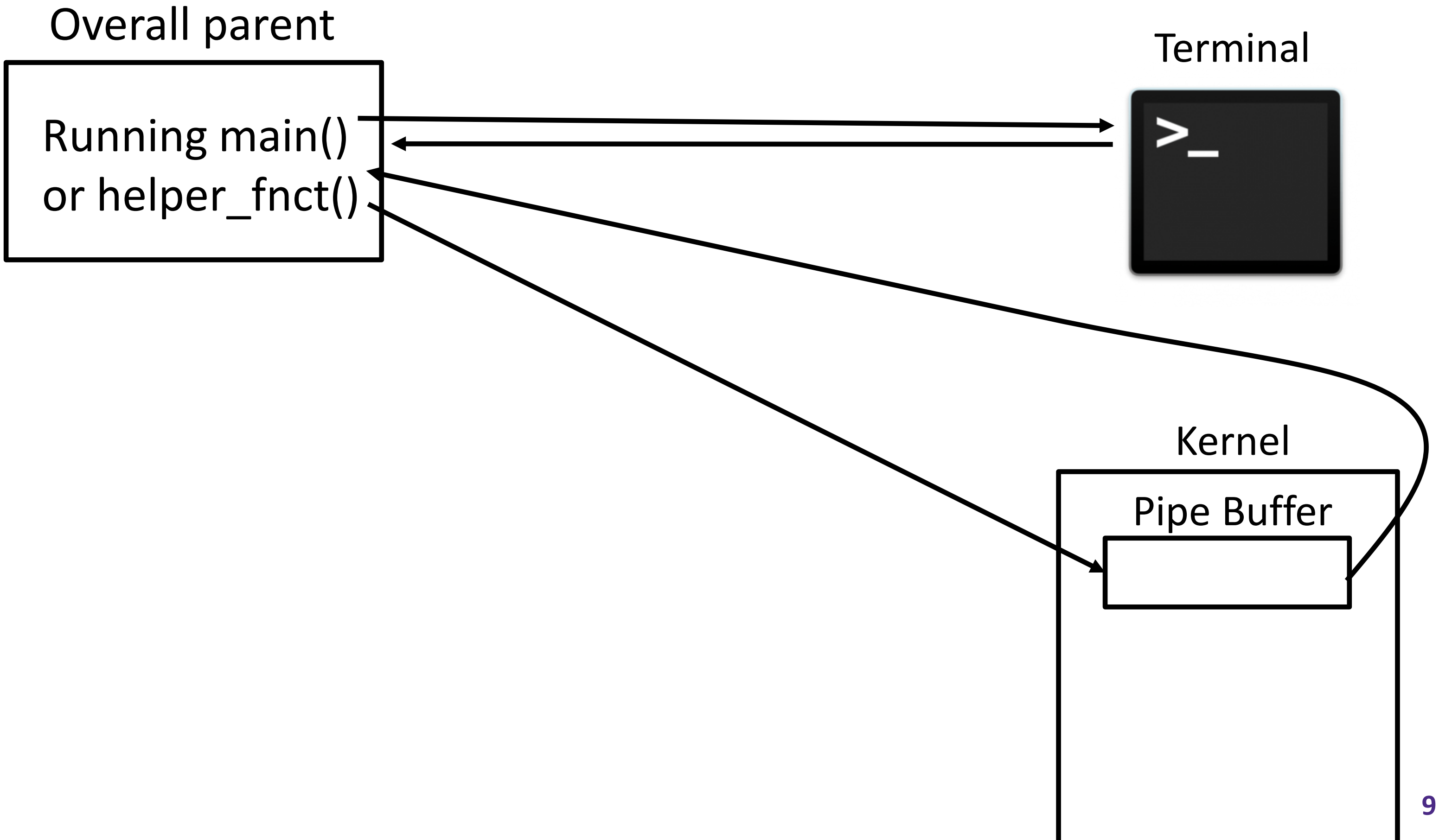
Pipe Example

- Take a look at `pipe_example.cpp` on the course website.
  - Simple pipe example

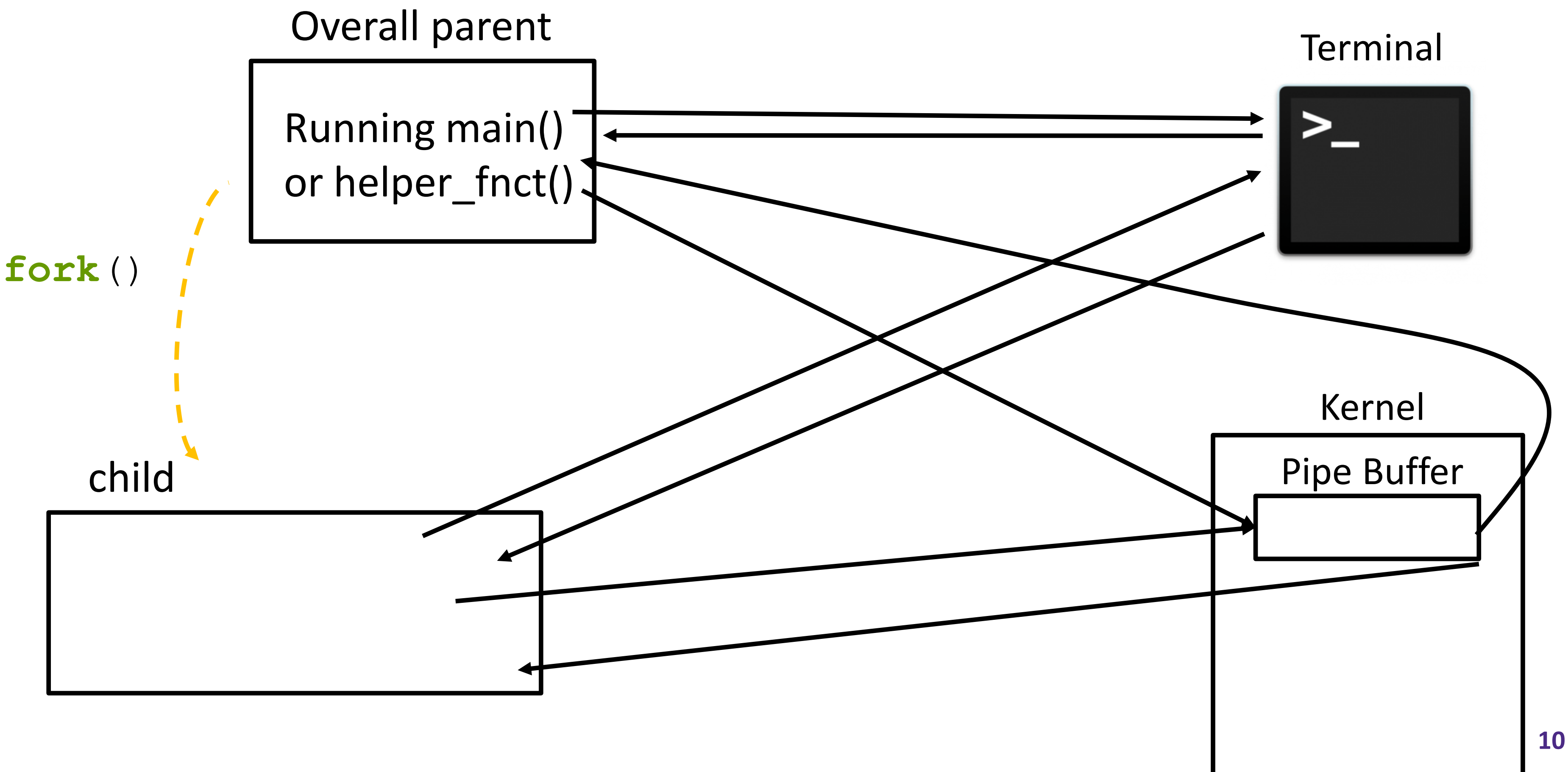
pipe_example.cpp Walkthrough



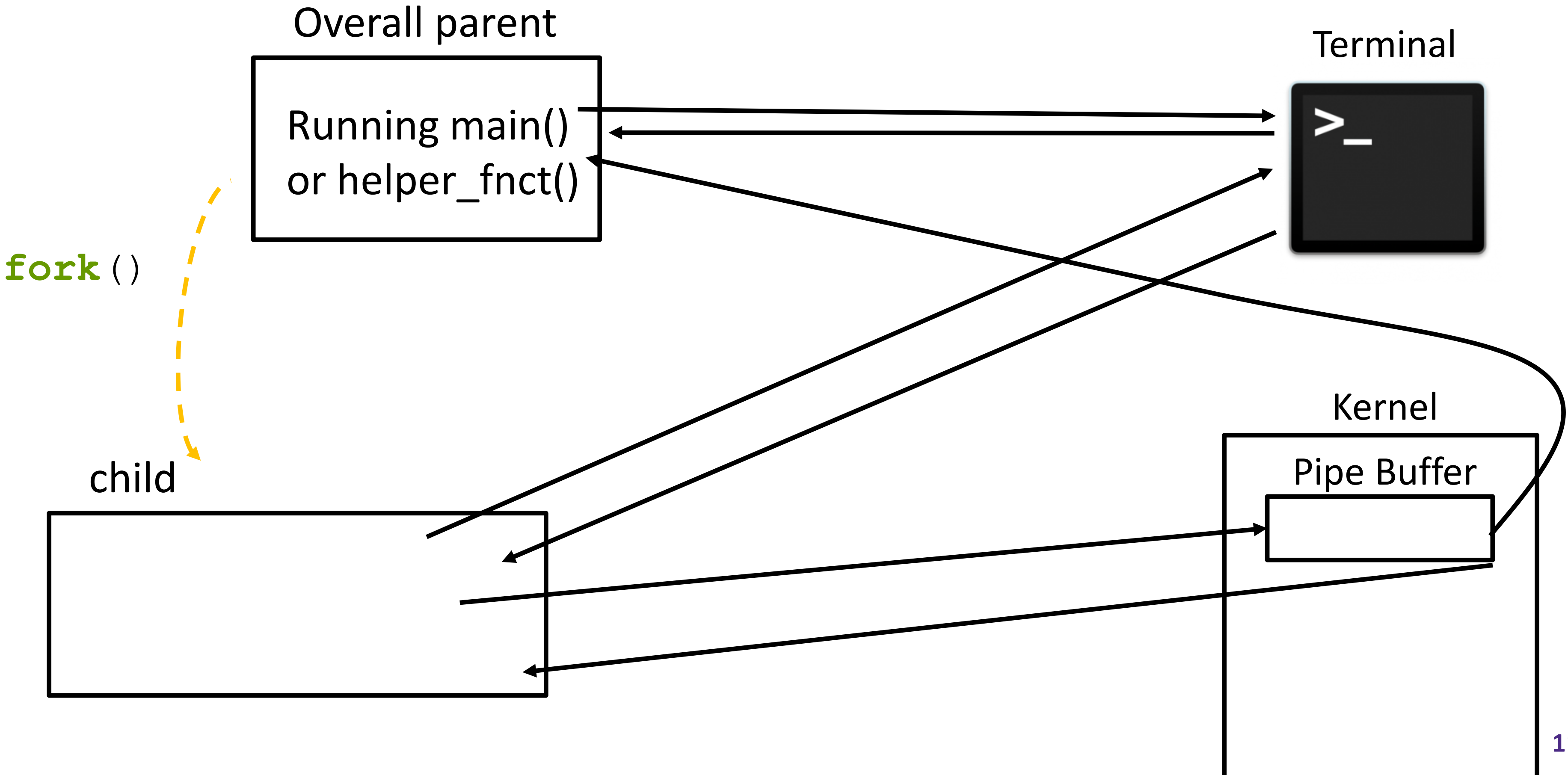
`pipe_example.cpp` Walkthrough



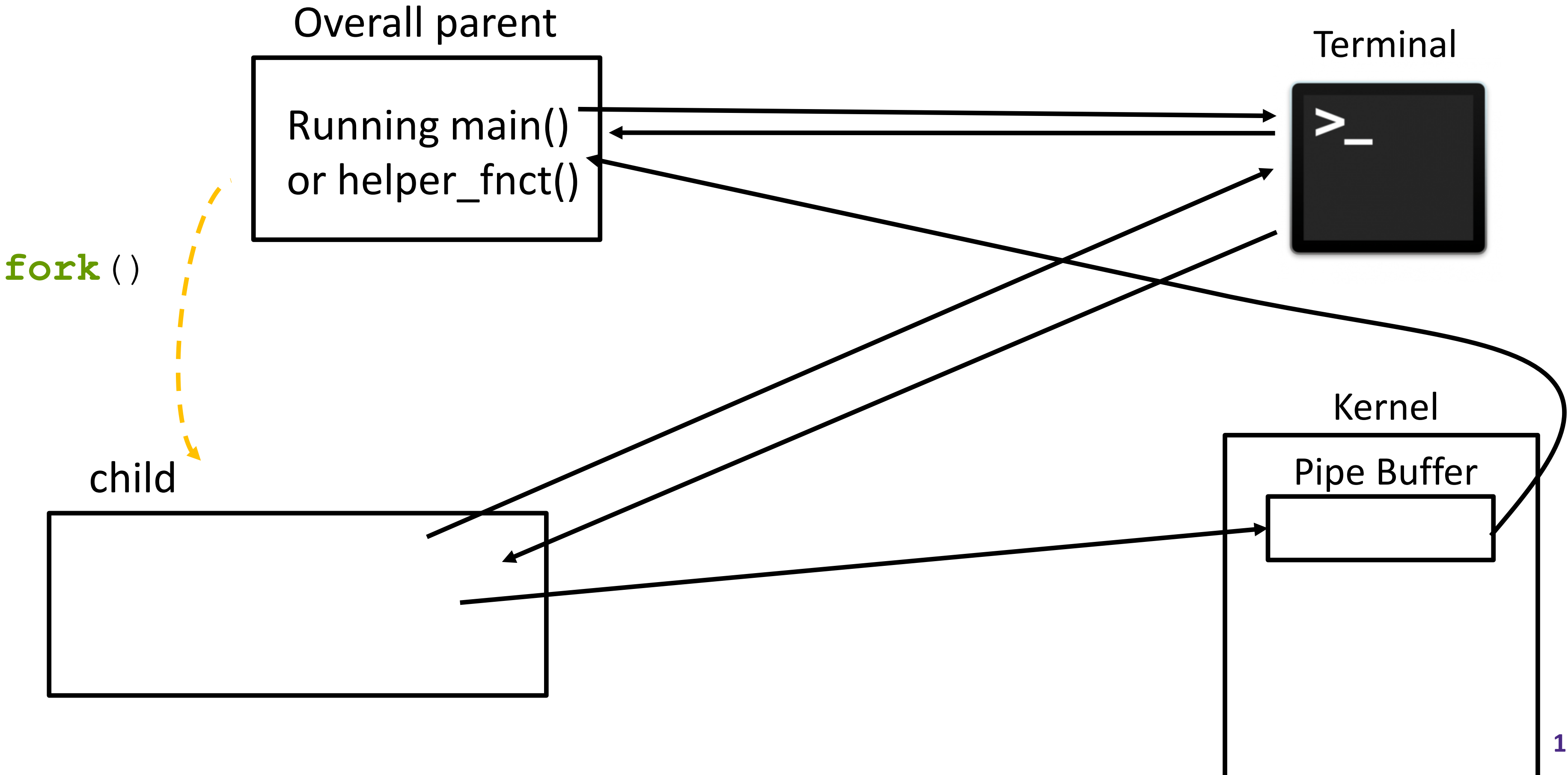
pipe_example.cpp Walkthrough



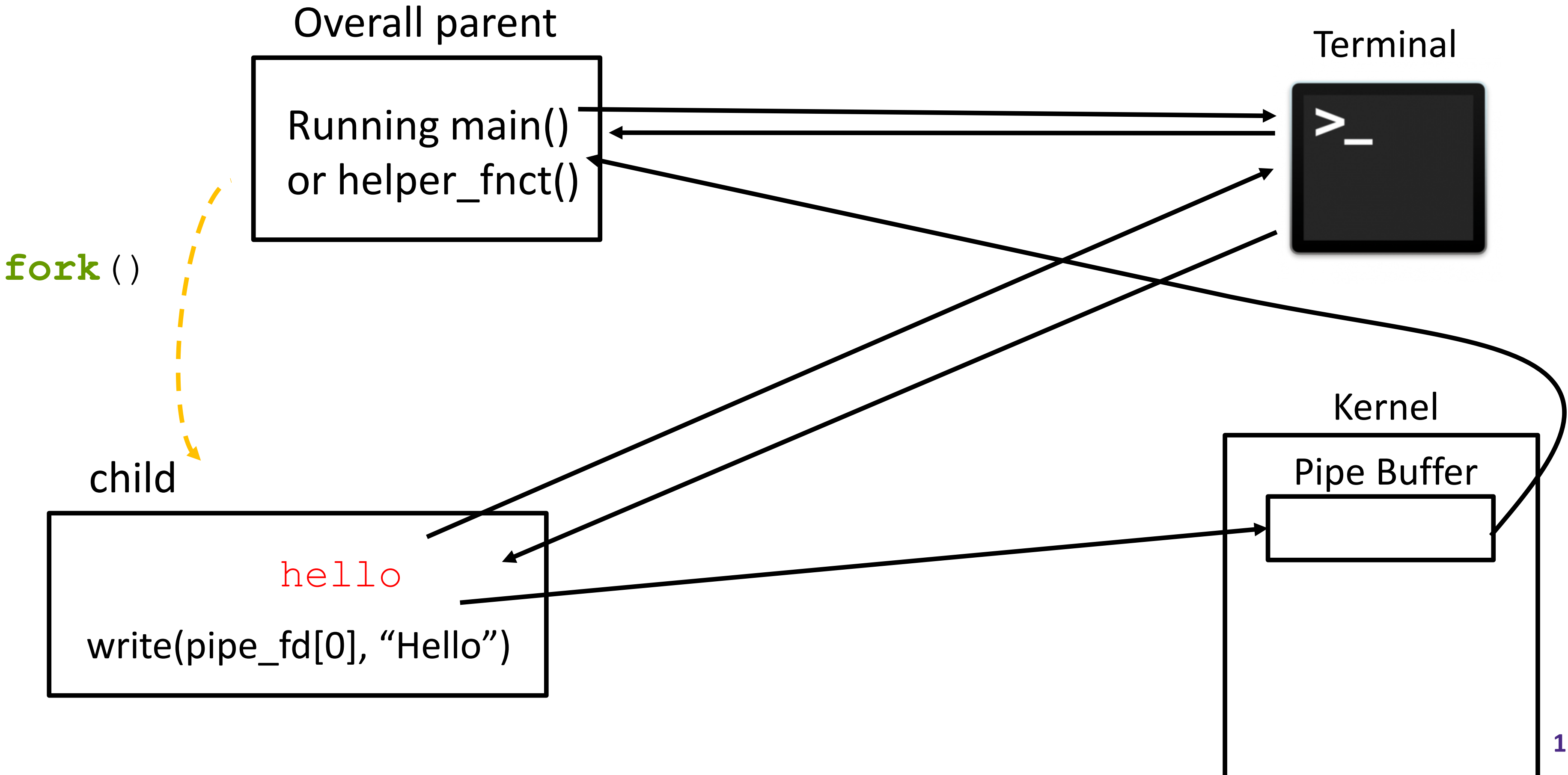
pipe_example.cpp Walkthrough



`pipe_example.cpp` Walkthrough

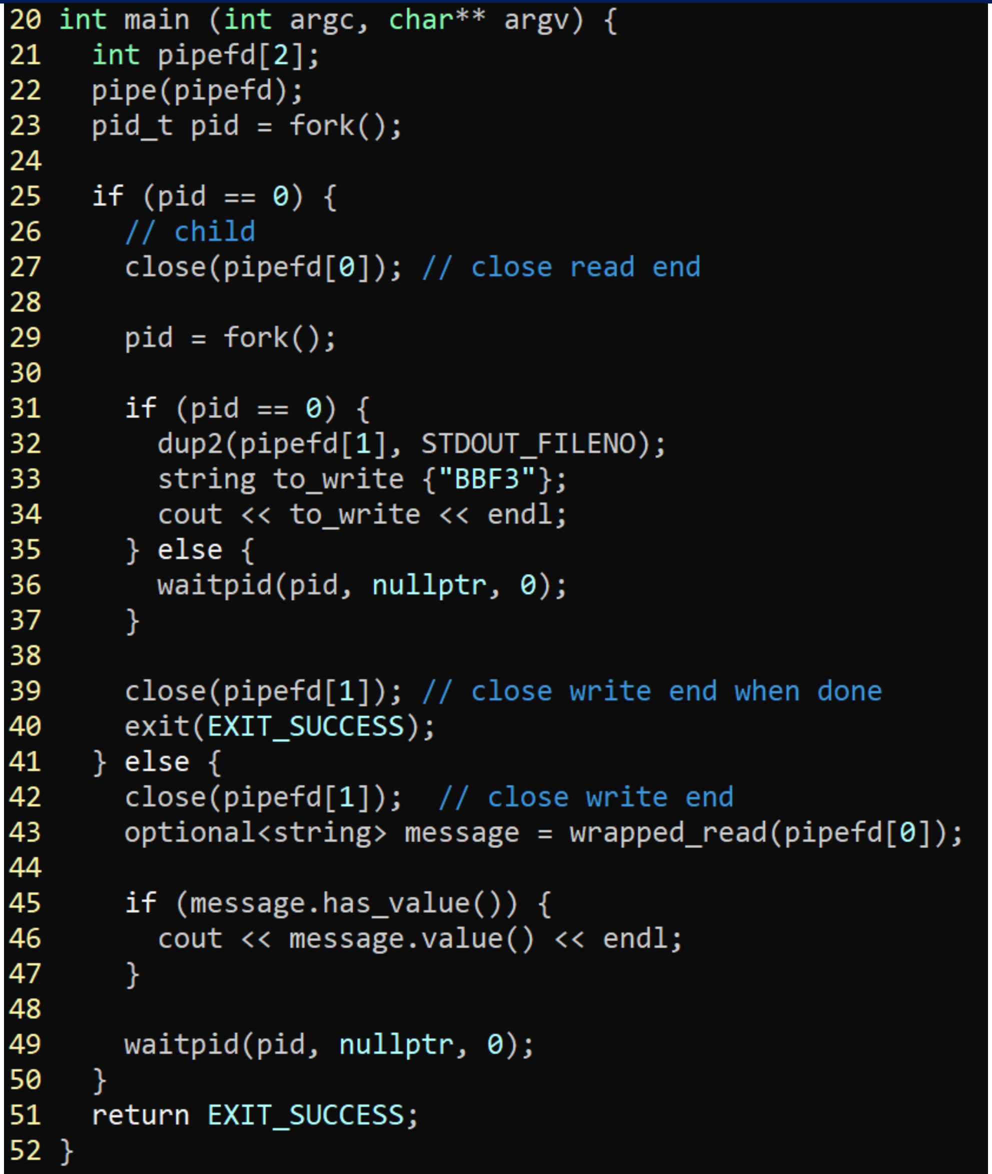


pipe_example.cpp Walkthrough



Poll Everywhere

pollev.com/tqm



```
int main (int argc, char** argv) {
  int pipefd[2];
  pipe(pipefd);
  pid_t pid = fork();

  if (pid == 0) {
    // child
    close(pipefd[0]); // close read end

    pid = fork();

    if (pid == 0) {
      dup2(pipefd[1], STDOUT_FILENO);
      string to_write {"BBF3"};
      cout << to_write << endl;
    } else {
      waitpid(pid, nullptr, 0);
    }

    close(pipefd[1]); // close write end when done
    exit(EXIT_SUCCESS);
  } else {
    close(pipefd[1]);  // close write end
    optional<string> message = wrapped_read(pipefd[0]);

    if (message.has_value()) {
      cout << message.value() << endl;
    }

    waitpid(pid, nullptr, 0);
  }
  return EXIT_SUCCESS;
}
```

- What does this code print? why? (assume pipe, close and fork succeed)
- **`pipe_poll.cpp`**

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

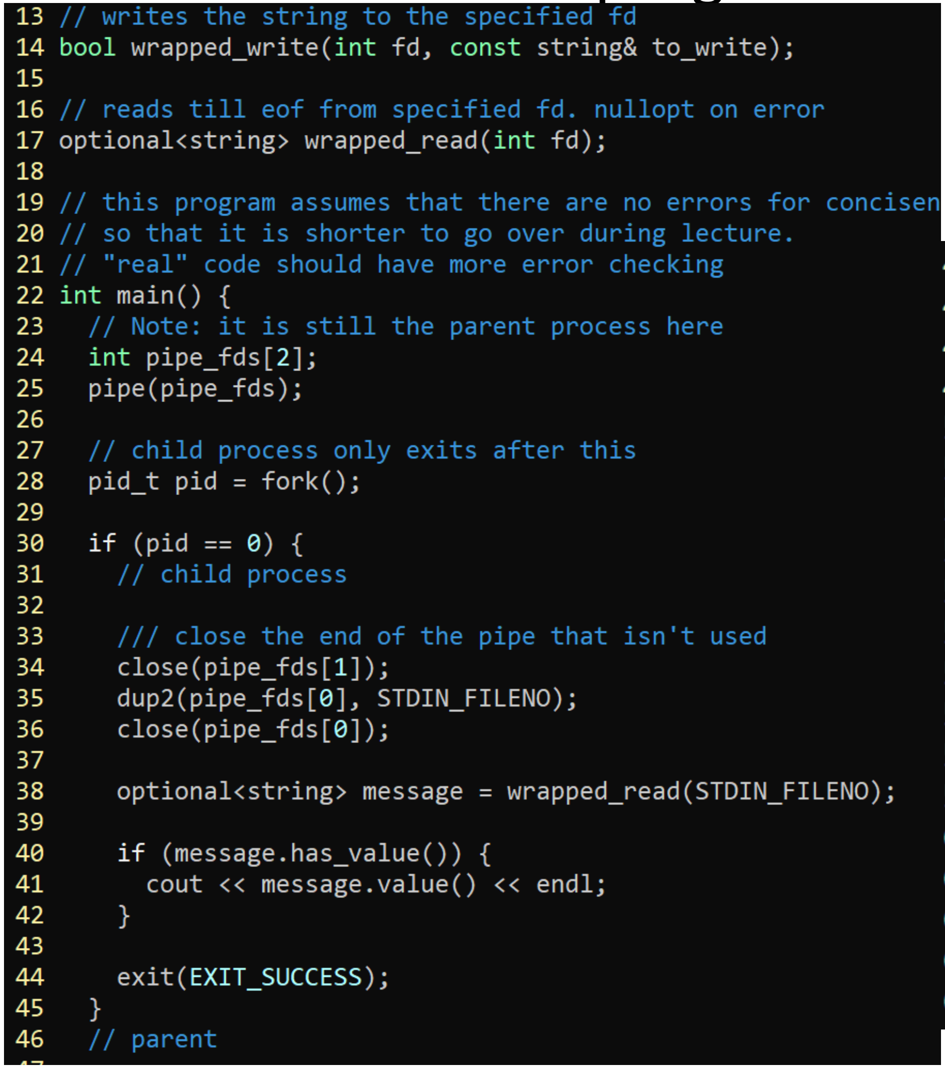
- **Good practice: CLOSE ALL PIPE FDS YOU ARE DONE WITH**

Poll Everywhere

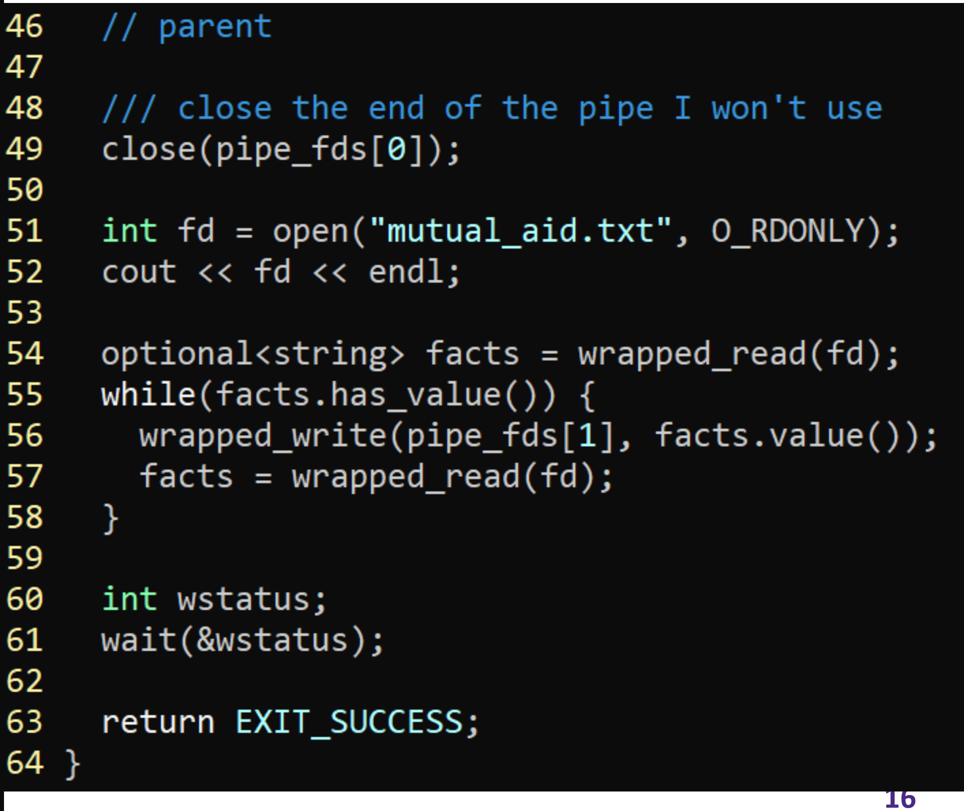
pollev.com/tqm

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

code is on website as **cat_pipe.cpp**



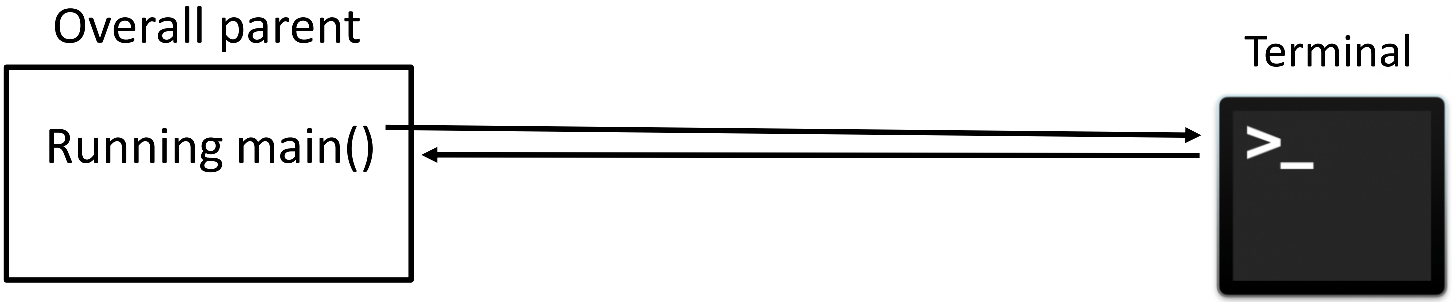
```
// writes the string to the specified fd
bool wrapped_write(int fd, const string& to_write);

// reads till eof from specified fd. nullopt on error
optional<string> wrapped_read(int fd);

// this program assumes that there are no errors for concisen
// so that it is shorter to go over during lecture.
// "real" code should have more error checking
int main() {
  // Note: it is still the parent process here
  int pipe_fds[2];
  pipe(pipe_fds);

  // child process only exits after this
  pid_t pid = fork();

  if (pid == 0) {
    // child process

    /// close the end of the pipe that isn't used
    close(pipe_fds[1]);
    dup2(pipe_fds[0], STDIN_FILENO);
    close(pipe_fds[0]);

    optional<string> message = wrapped_read(STDIN_FILENO);

    if (message.has_value()) {
      cout << message.value() << endl;
    }

    exit(EXIT_SUCCESS);
  }
  // parent
```



```
  // parent

  /// close the end of the pipe I won't use
  close(pipe_fds[0]);

  int fd = open("mutual_aid.txt", O_RDONLY);
  cout << fd << endl;

  optional<string> facts = wrapped_read(fd);
  while(facts.has_value()) {
    wrapped_write(pipe_fds[1], facts.value());
    facts = wrapped_read(fd);
  }

  int wstatus;
  wait(&wstatus);

  return EXIT_SUCCESS;
}
```

`cat_pipe.cpp` Trace

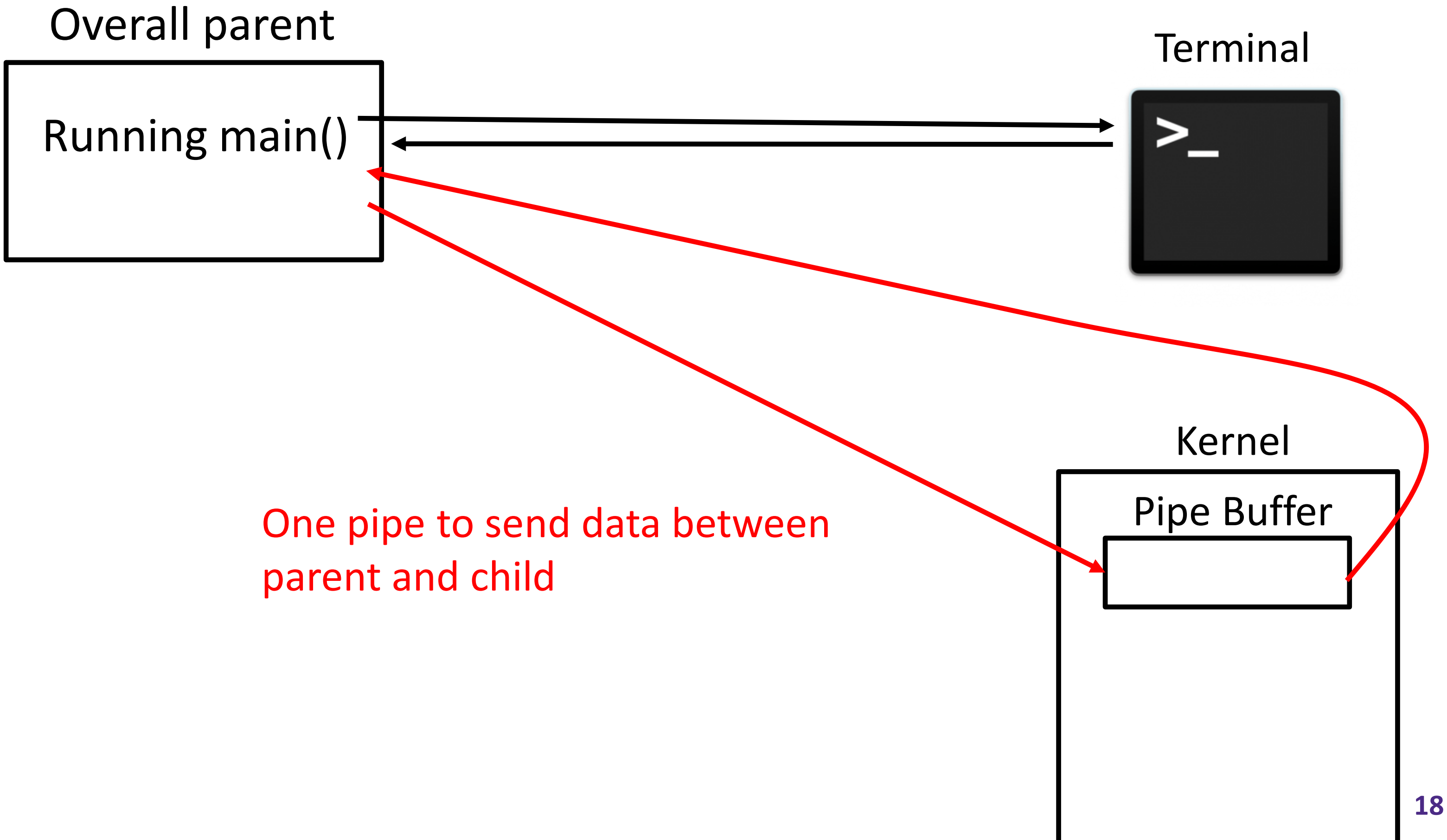
- First:
  we create a pipe



`cat_pipe.cpp` Trace

- First:

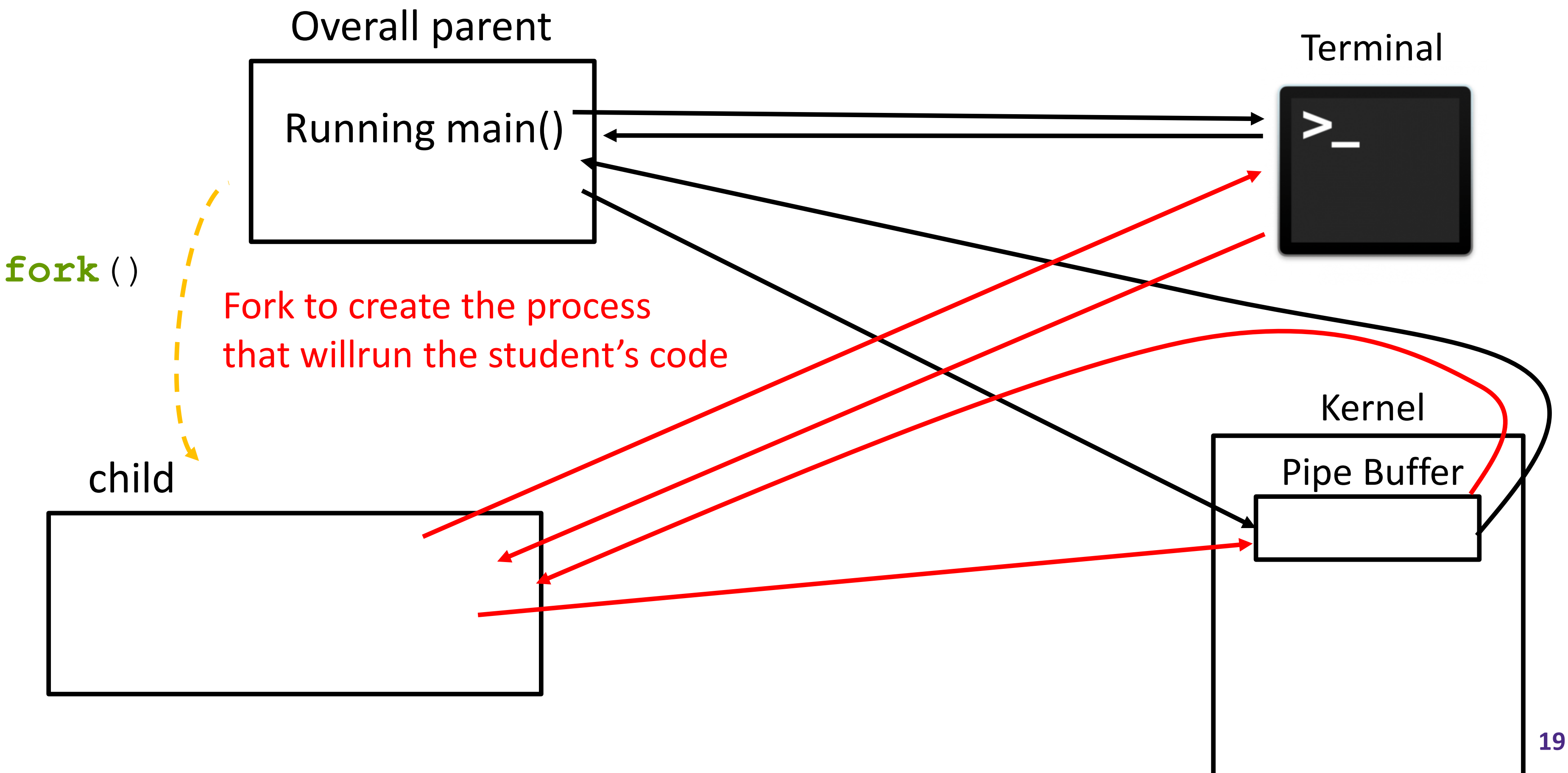
  we create a pipe



One pipe to send data between parent and child

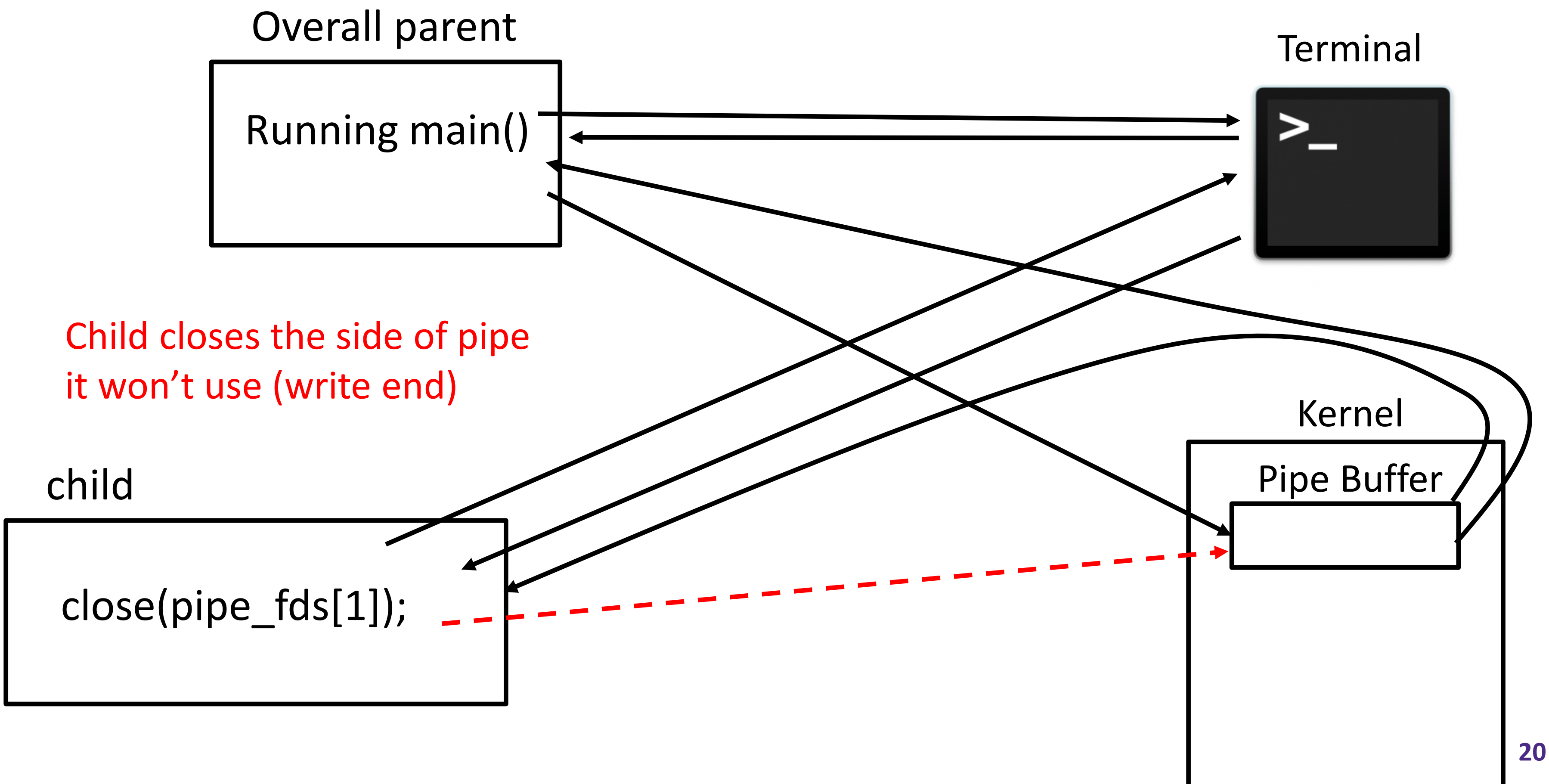
`cat_pipe.cpp` Trace

- second
  Fork



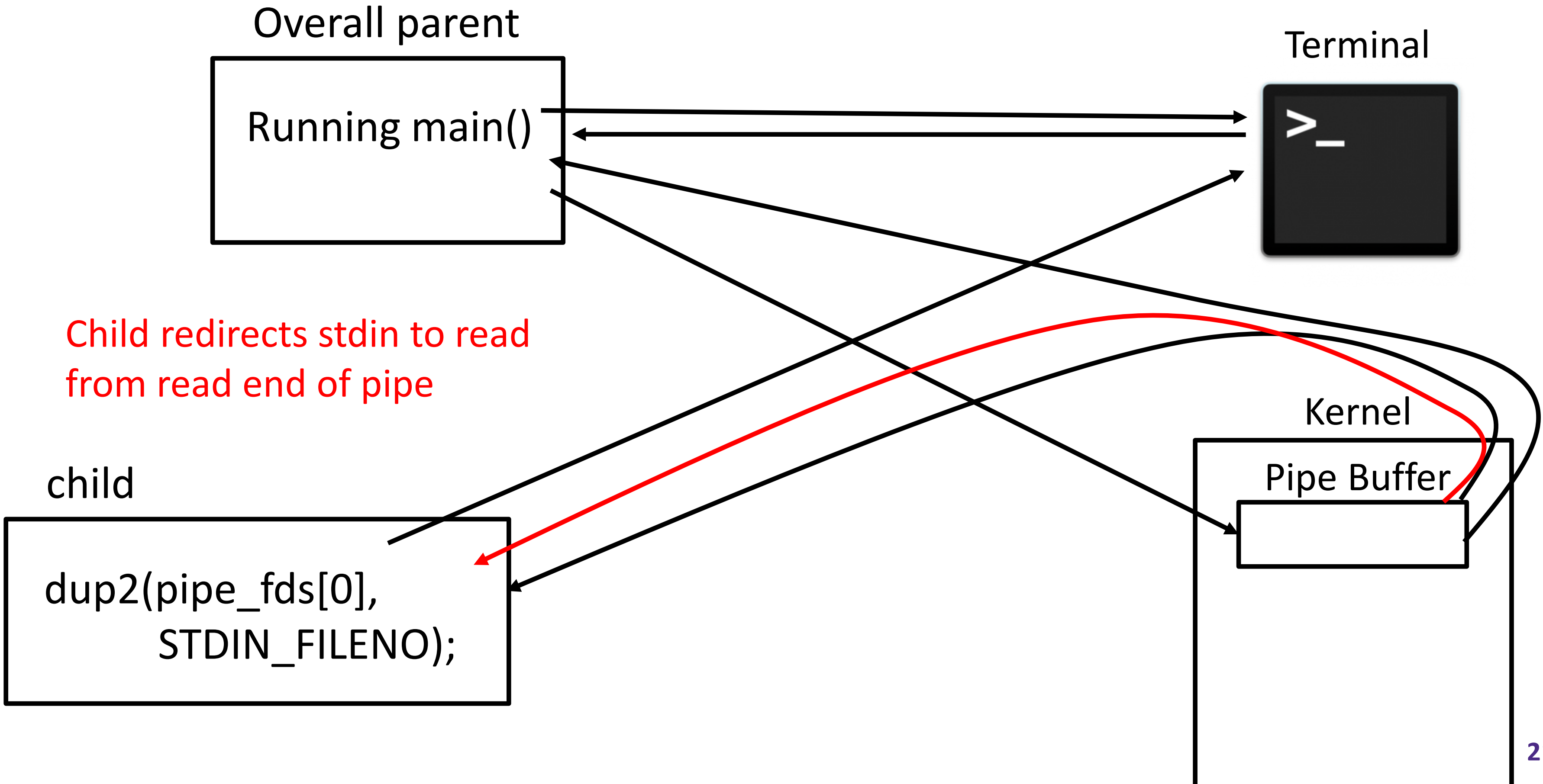
cat_pipe.cpp Trace

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



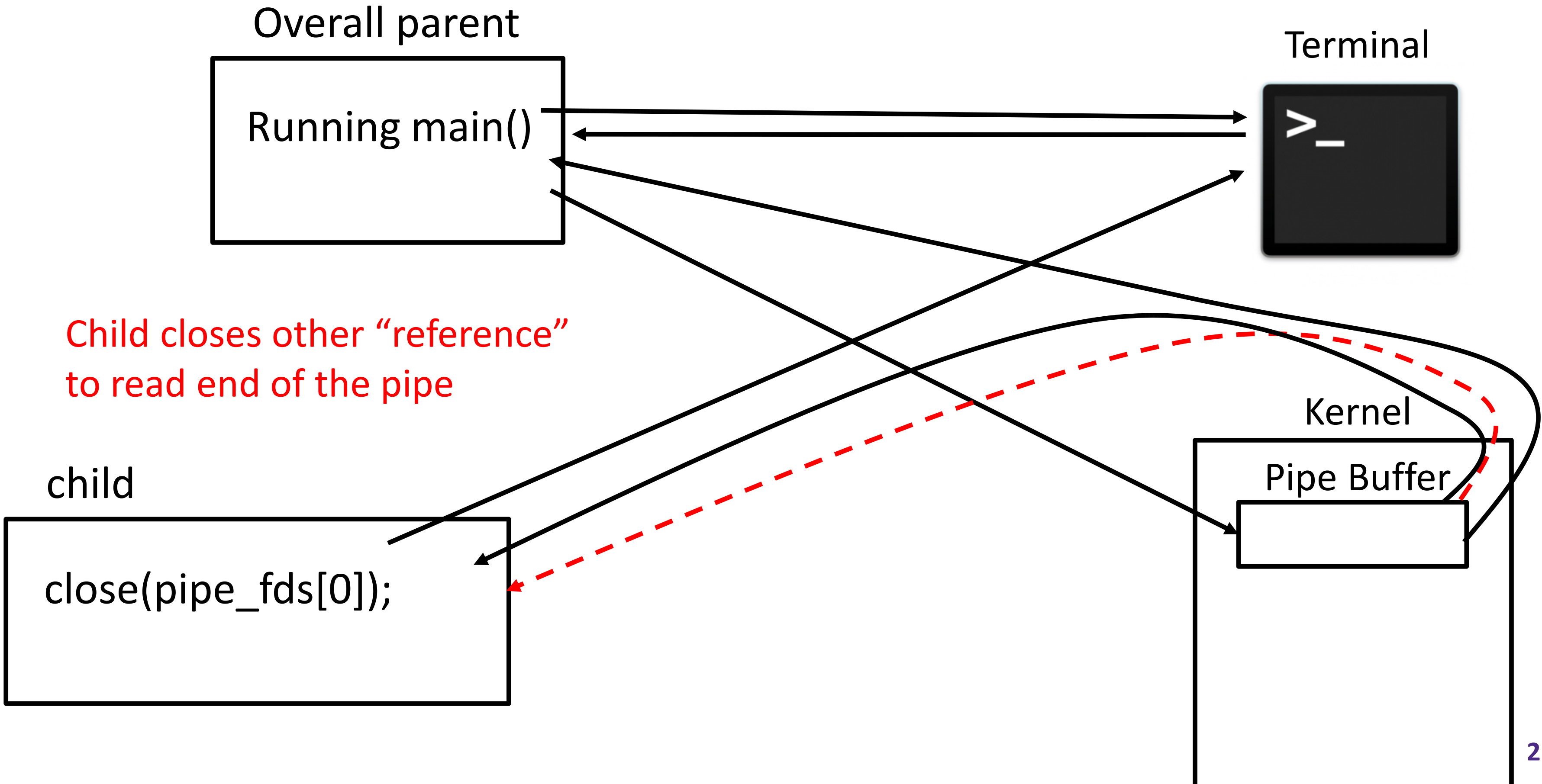
`cat_pipe.cpp` Trace

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



`cat_pipe.cpp` Trace

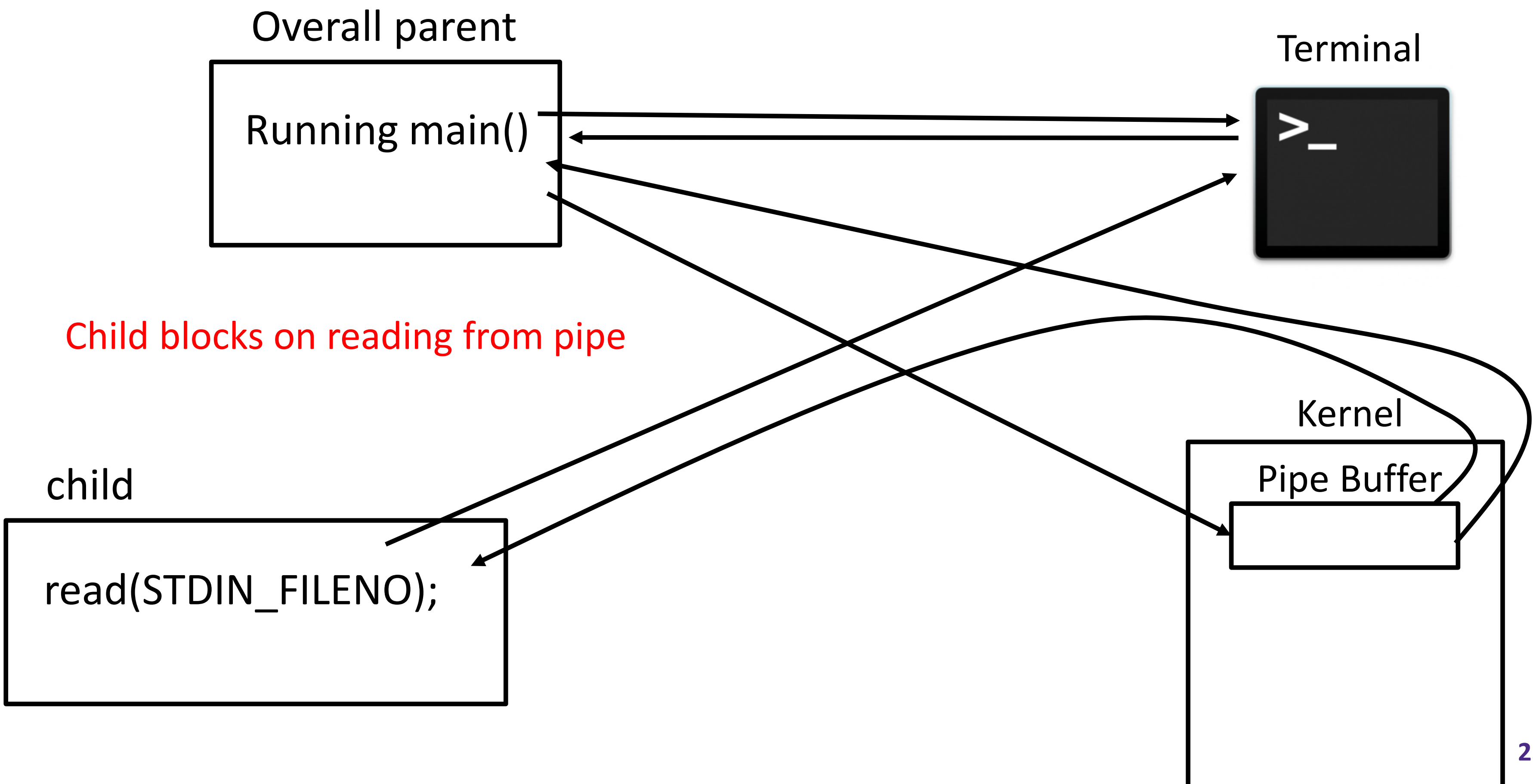
- Walking through child, but parent could be running first, after, or at the same time



Child closes other "reference" to read end of the pipe

`cat_pipe.cpp` Trace

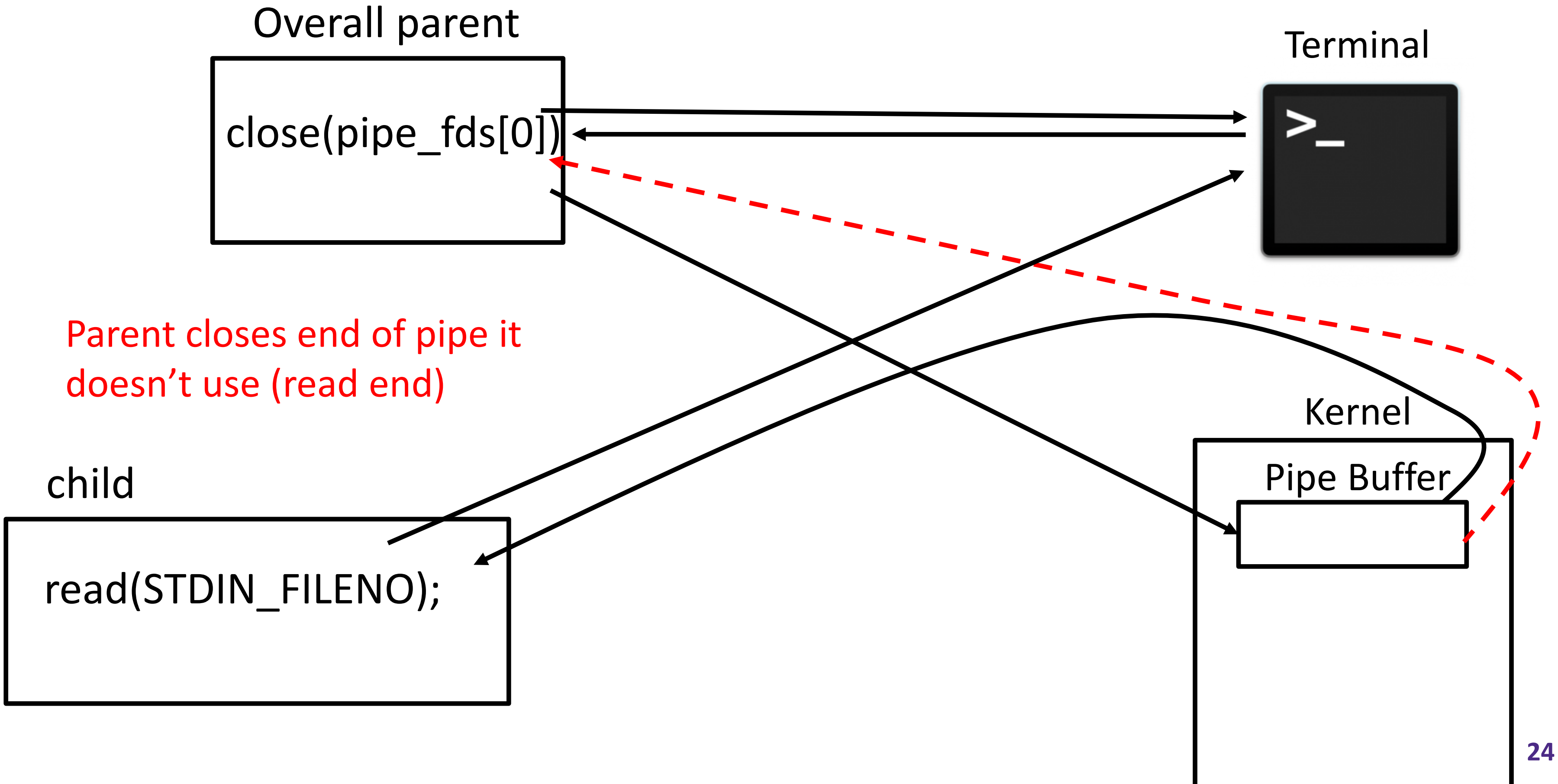
- Walking through child, but parent could be running first, after, or at the same time



Child blocks on reading from pipe

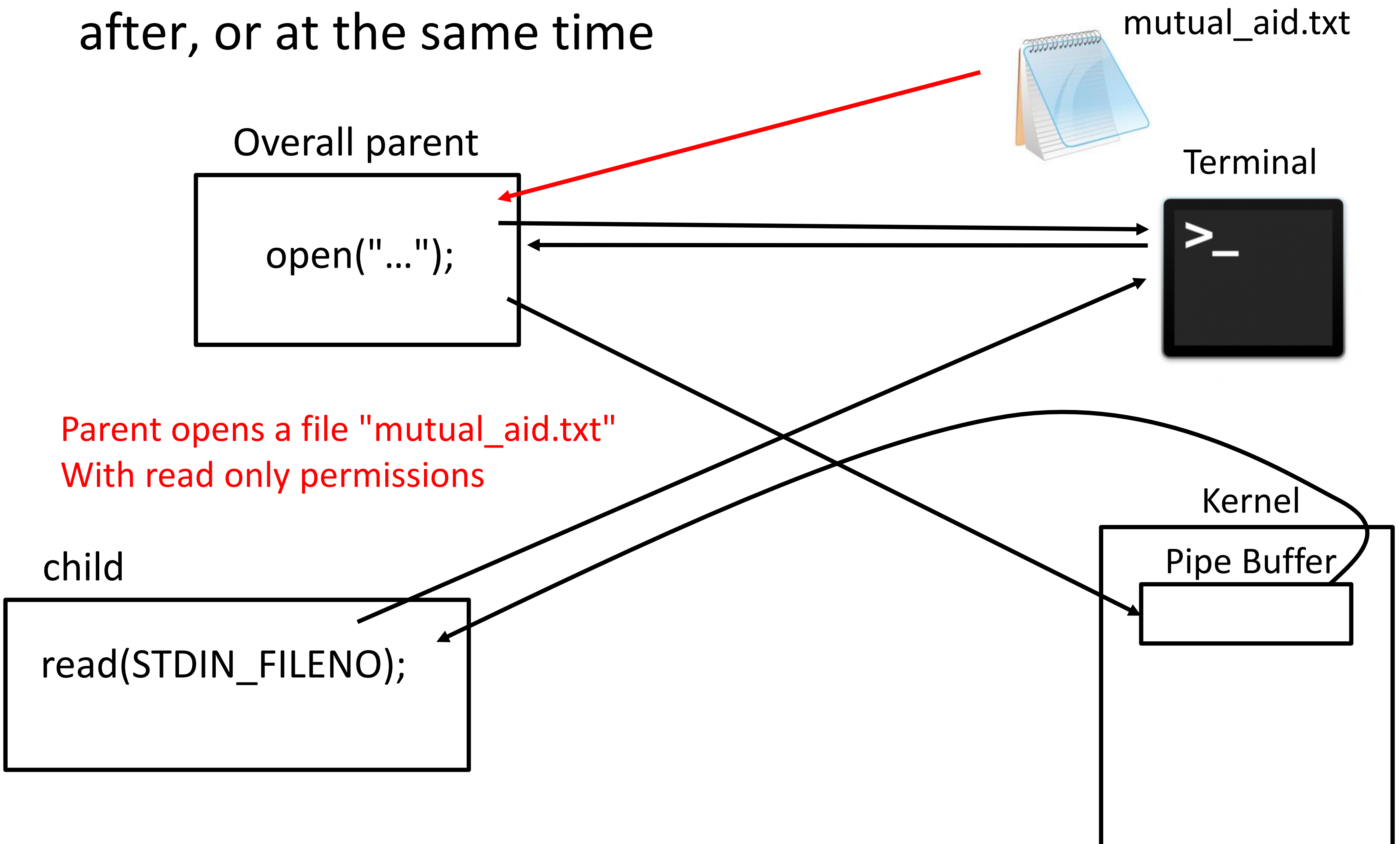
`cat_pipe.cpp` Trace

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



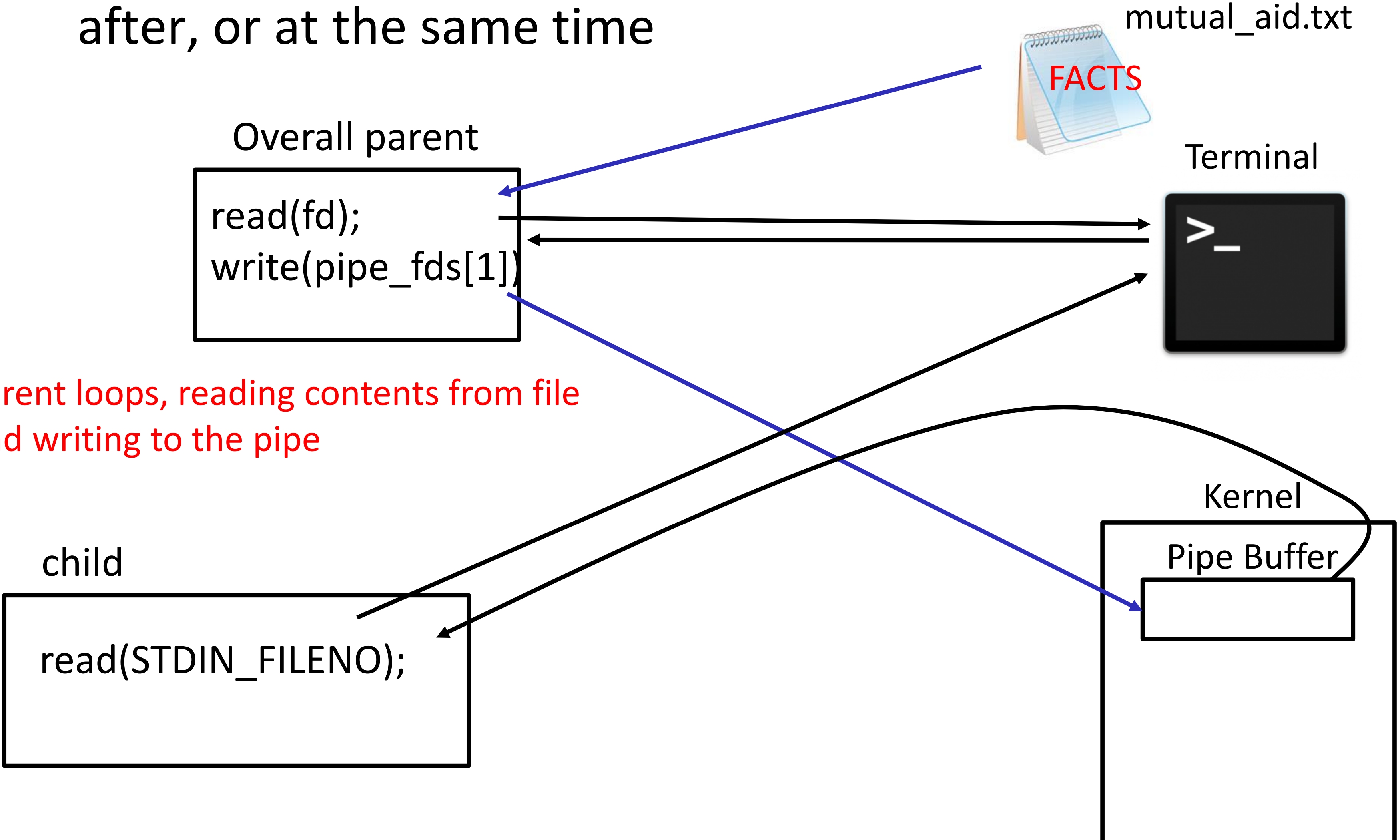
`cat_pipe.cpp` Trace

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



`cat_pipe.cpp` Trace

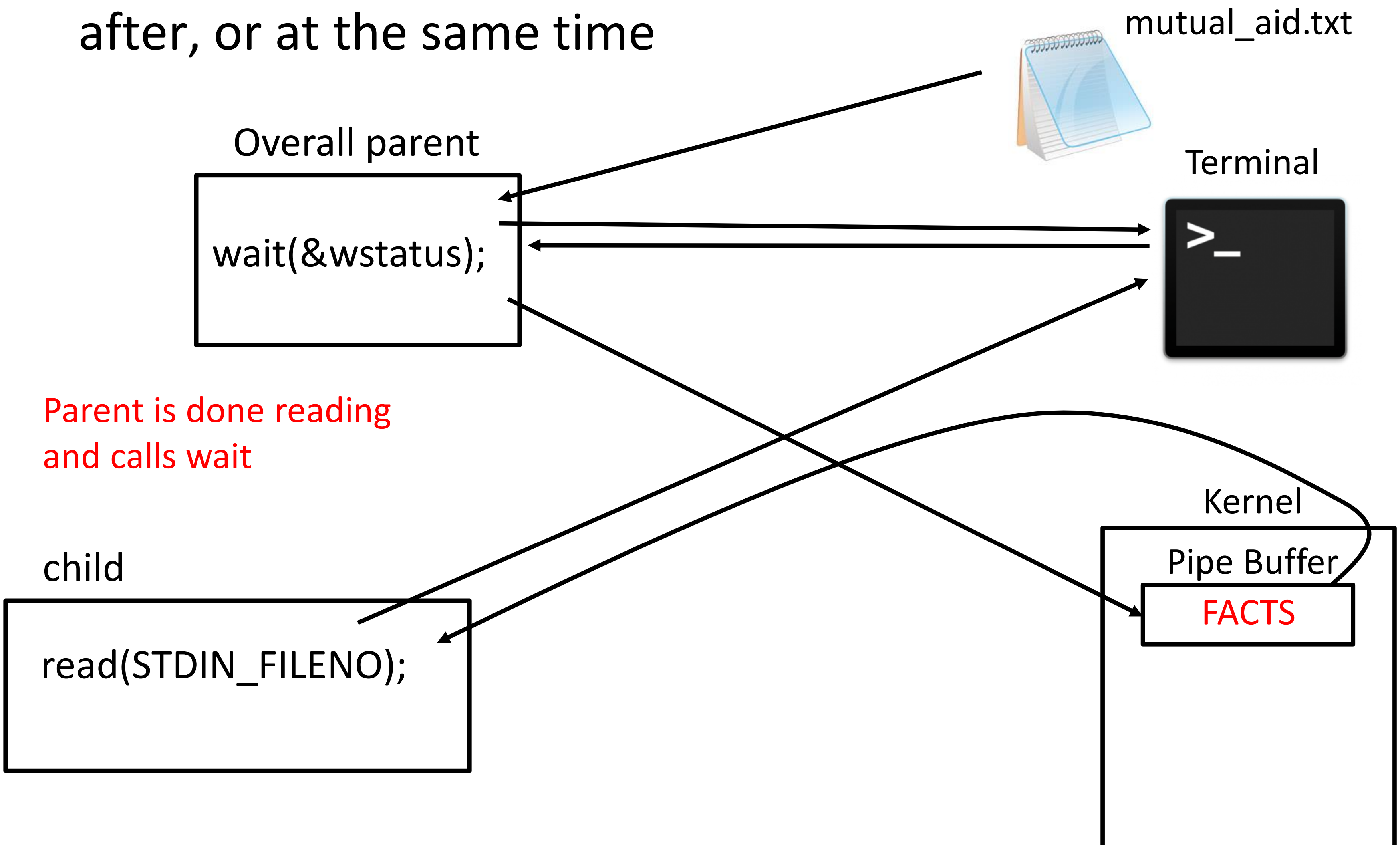
- Walking through **parent**, but child could be running first, after, or at the same time



Parent loops, reading contents from file and writing to the pipe

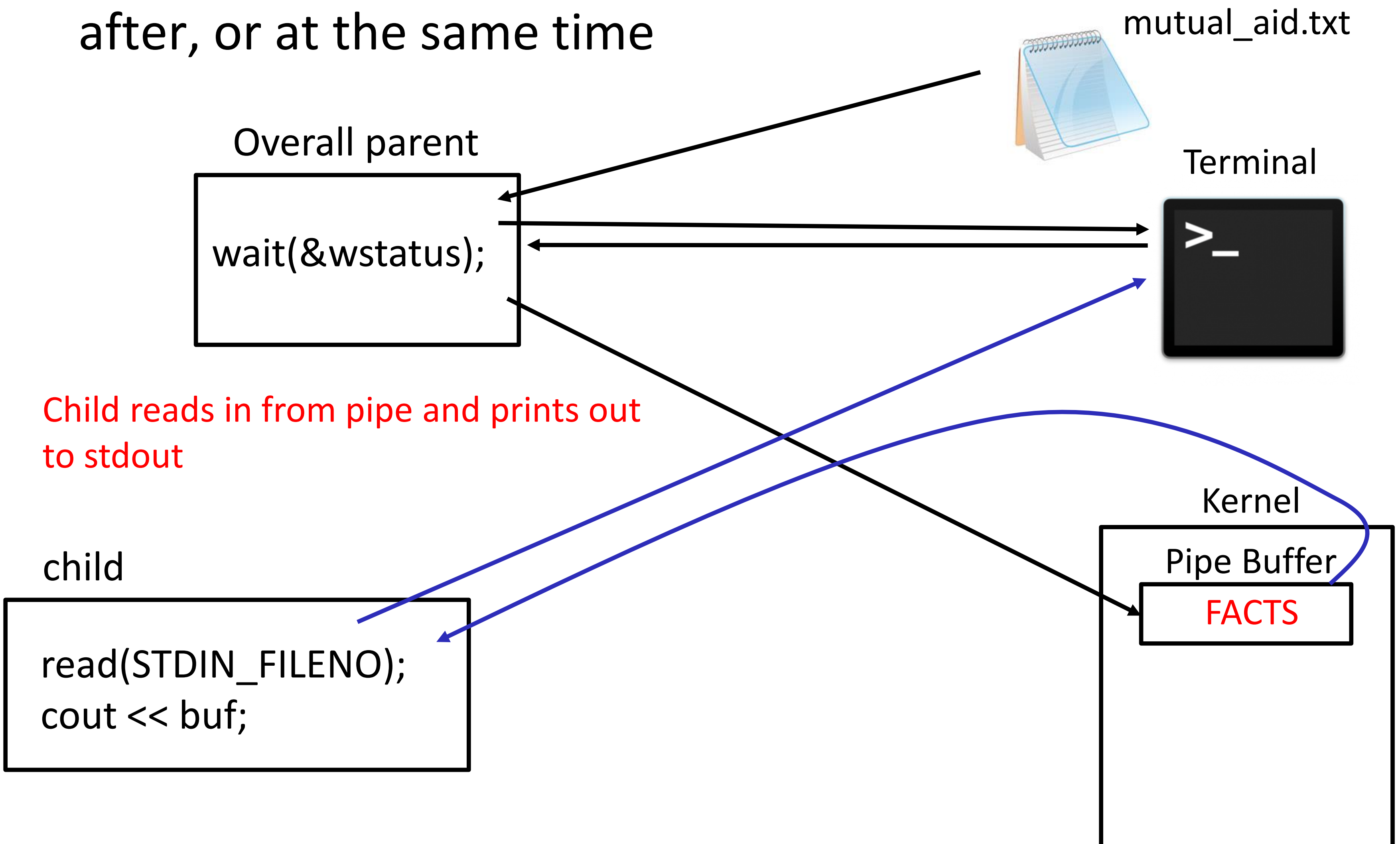
cat_pipe.cpp Trace

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



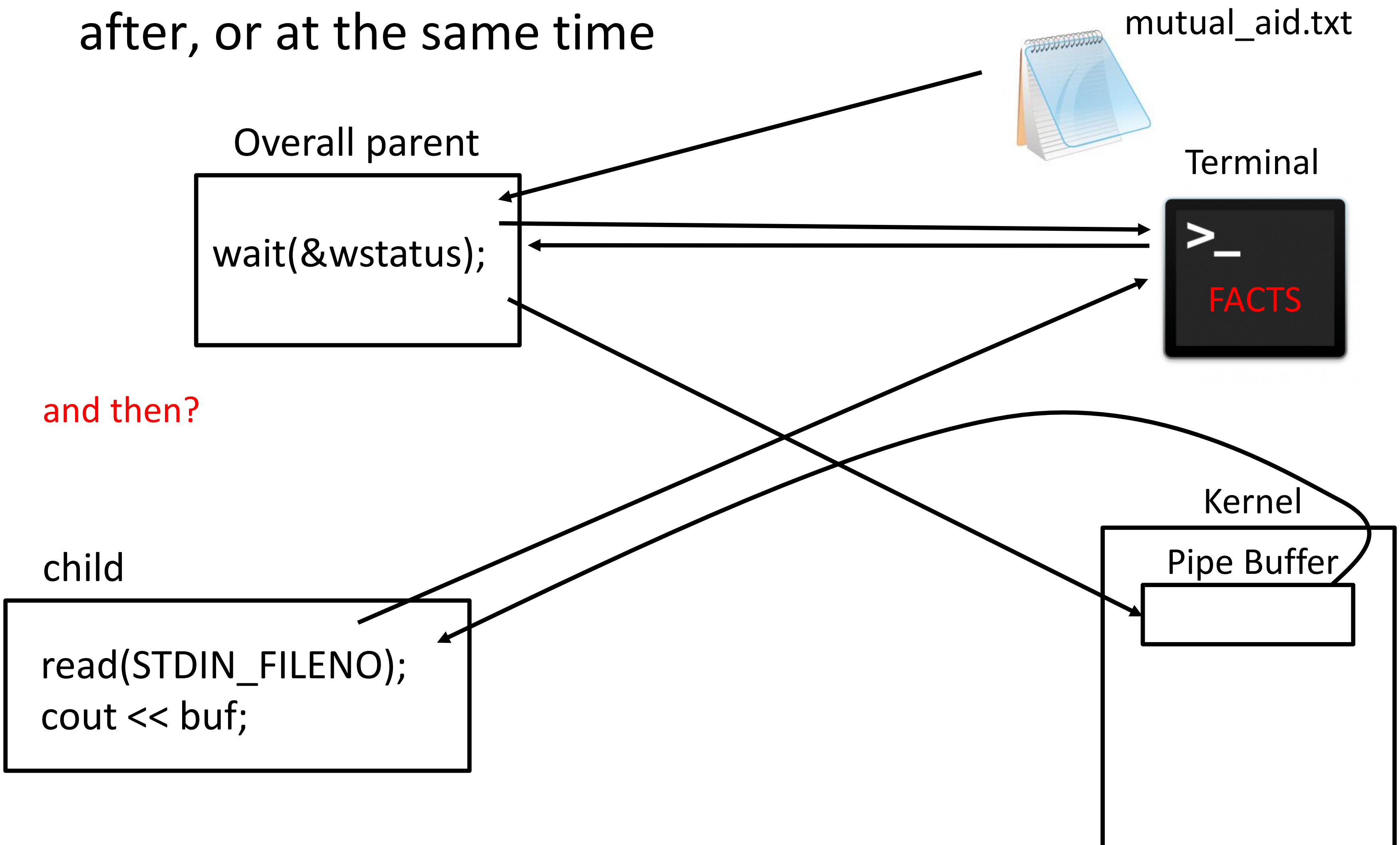
`cat_pipe.cpp` Trace

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



`cat_pipe.cpp` Trace

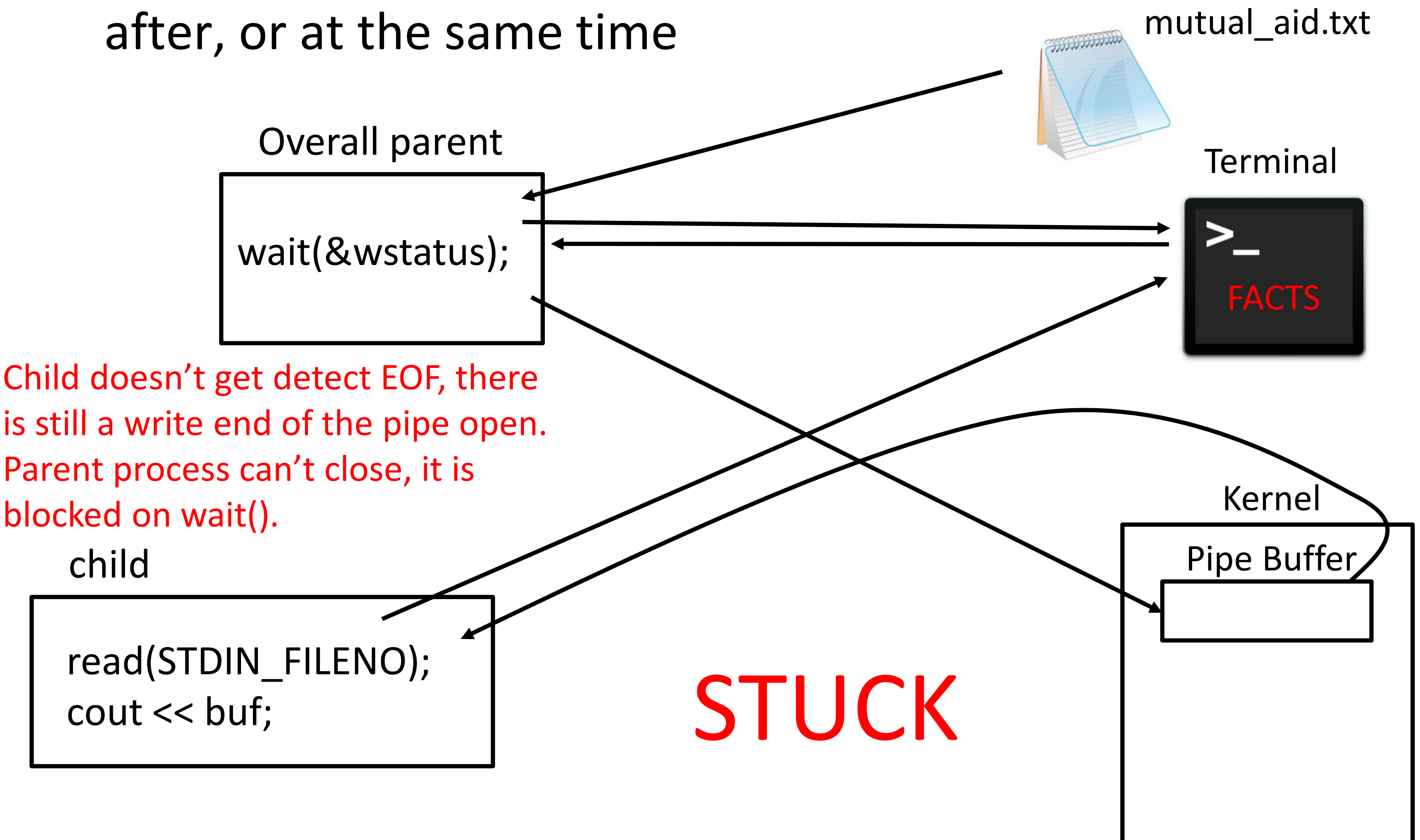
- Walking through **parent**, but child could be running first, after, or at the same time



and then?

cat_pipe.cpp Trace

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



Child doesn't get detect EOF, there is still a write end of the pipe open. Parent process can't close, it is blocked on wait().

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**

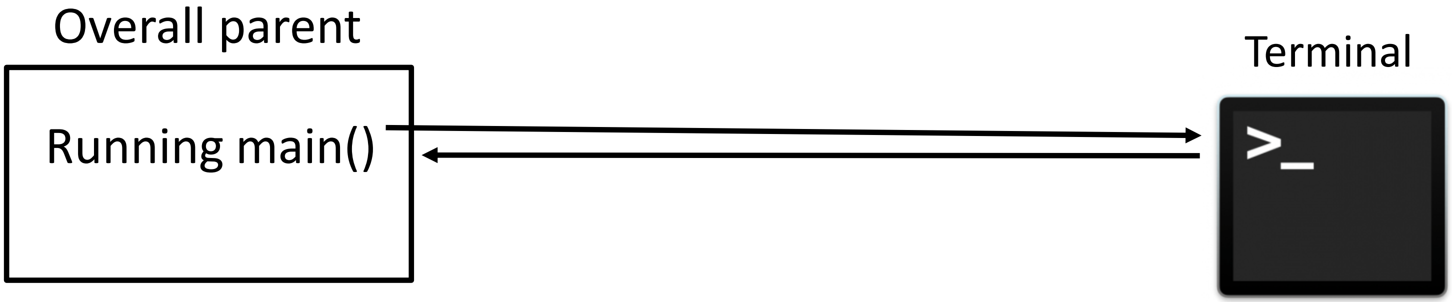
Exec & Pipe Demo

- See `autograder.cpp`
  - Example of using exec and fork

- See `io_autograder.cpp`
  - 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.cpp` Trace

- First:
  we compile the program with the gcc command



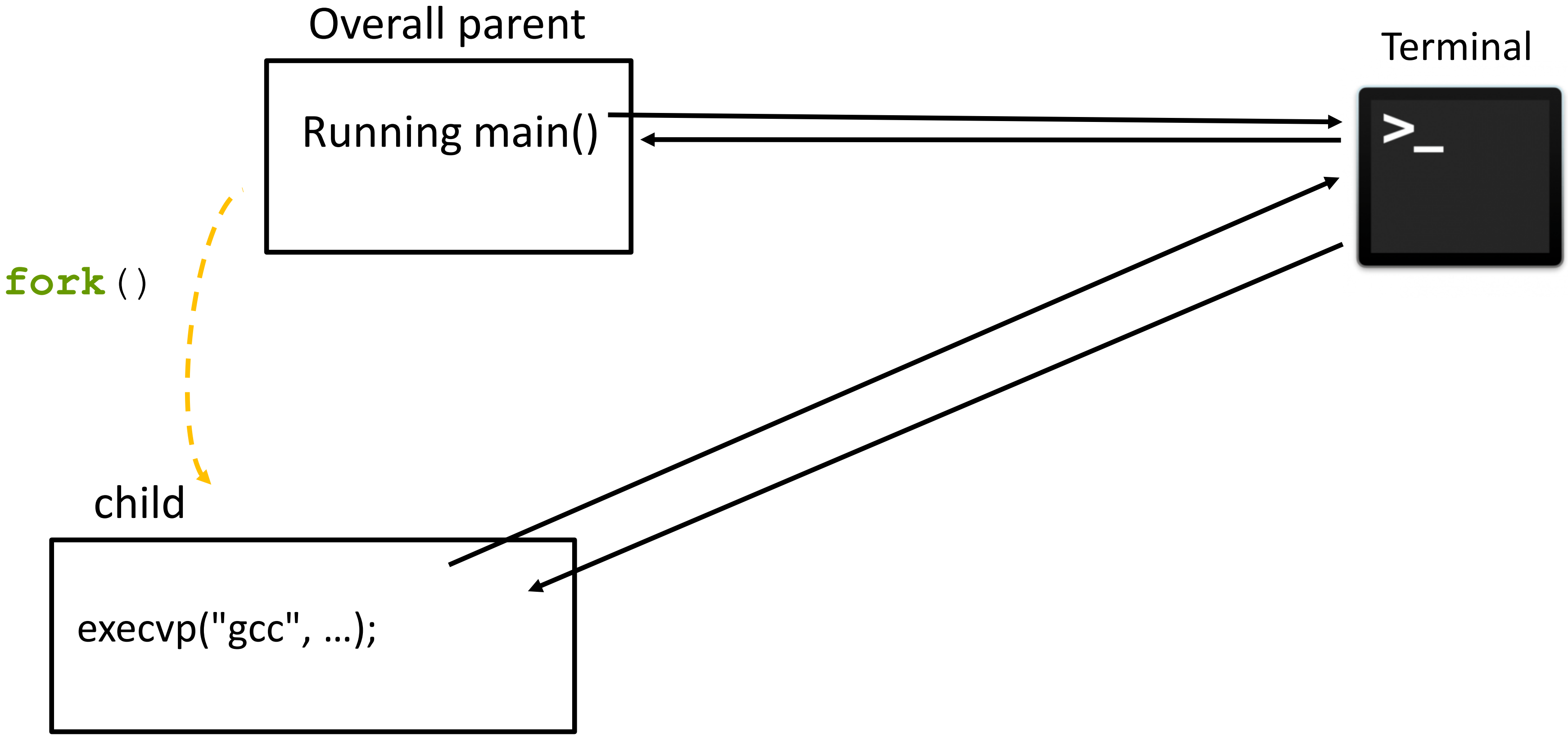
Overall parent

Running main()

Terminal

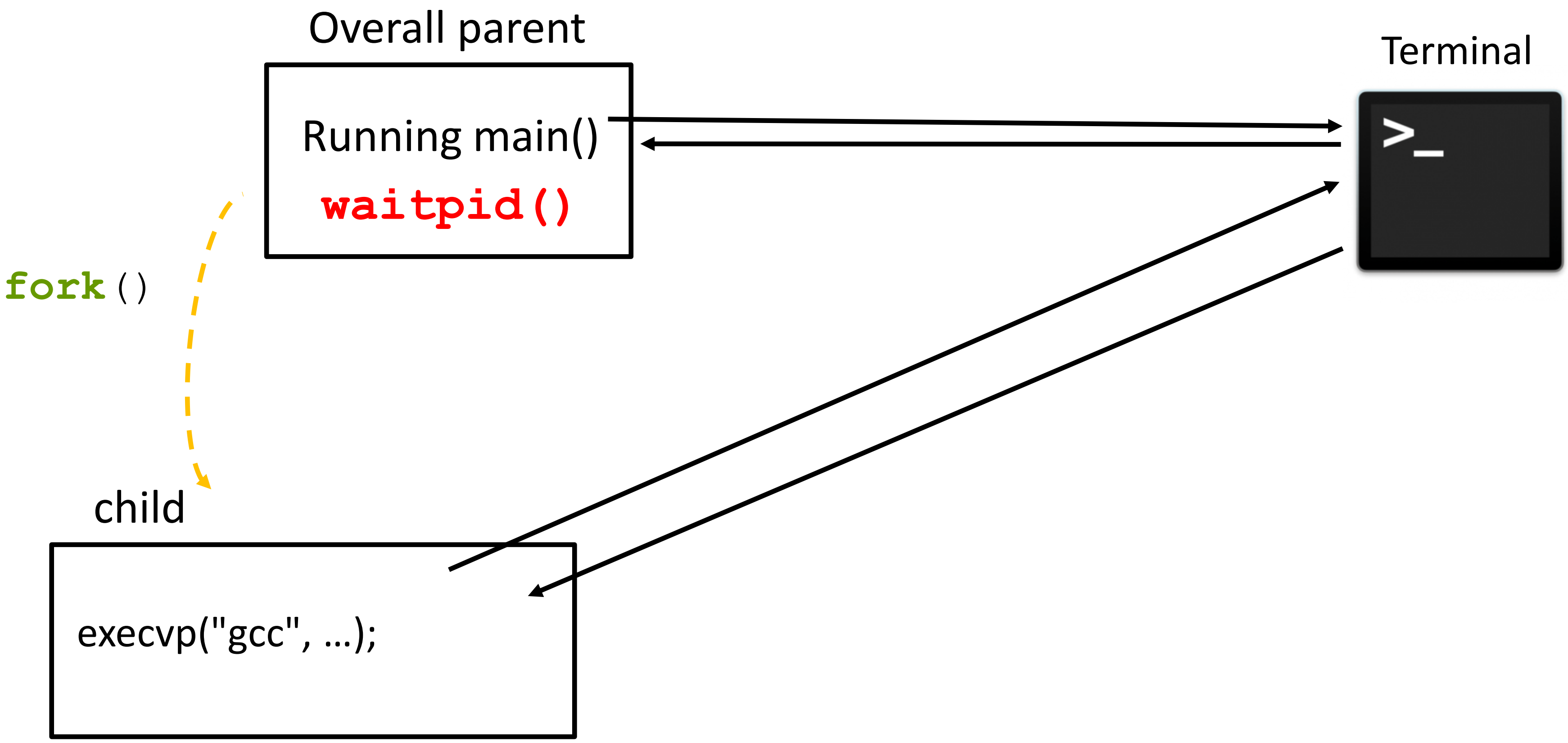
`io_autograder.cpp` Trace

- First:
  we compile the program with the gcc command



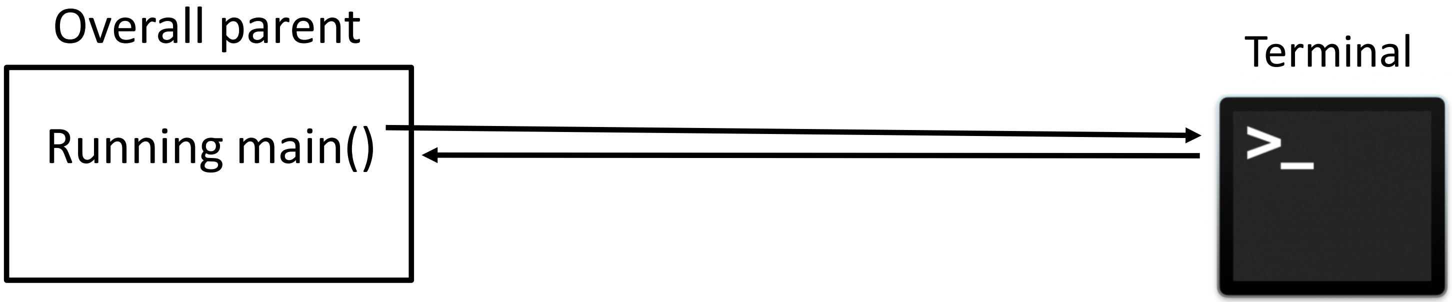
io_autograder.cpp Trace

- First:
  we compile the program with the gcc command



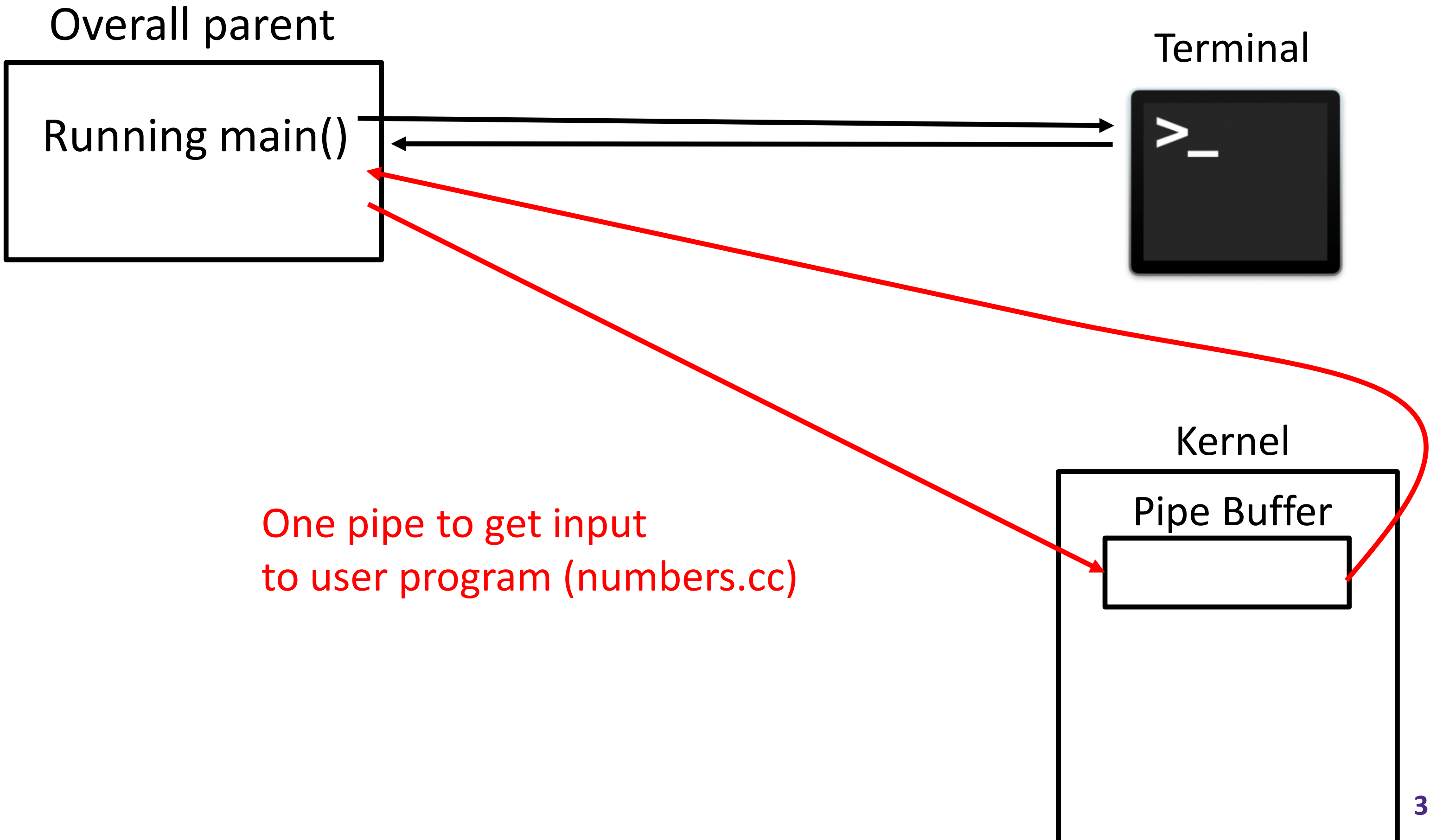
`io_autograder.cpp` Trace

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



`io_autograder.cpp` Trace

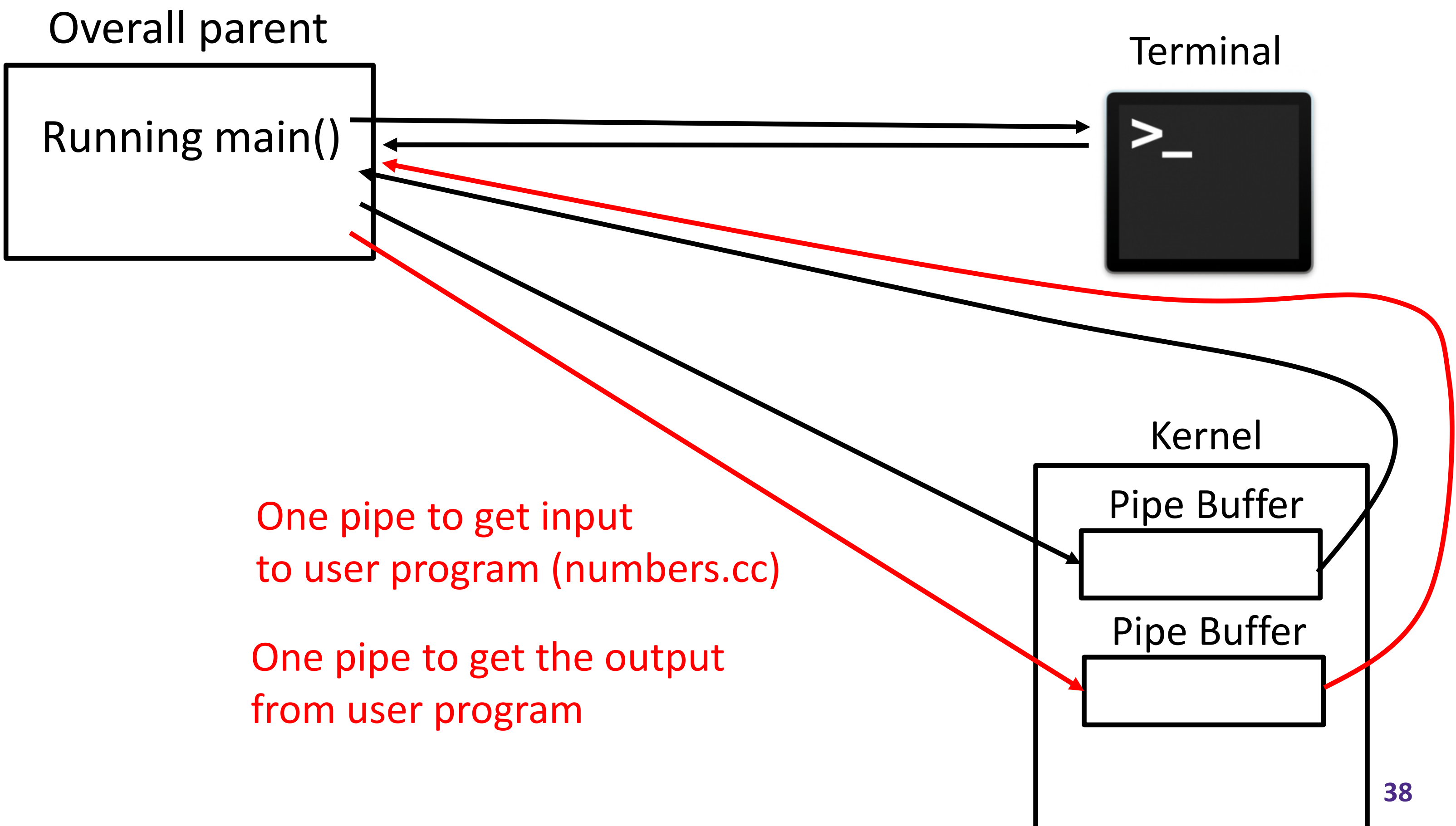
- Compilation done! Run the compiled program… BUT send autograder input and capture output



One pipe to get input to user program (numbers.cc)

`io_autograder.cpp` Trace

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

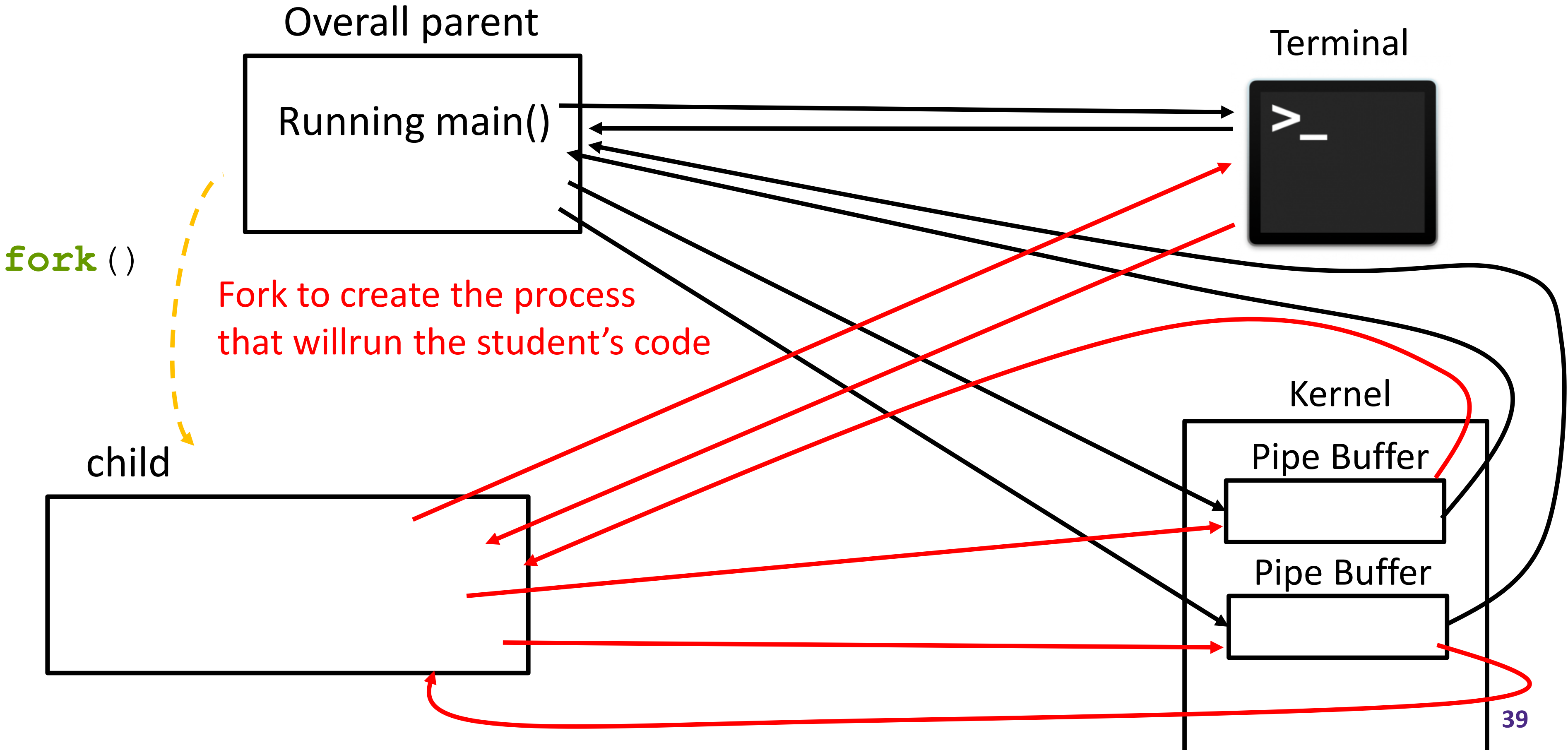


One pipe to get input to user program (numbers.cc)

One pipe to get the output from user program

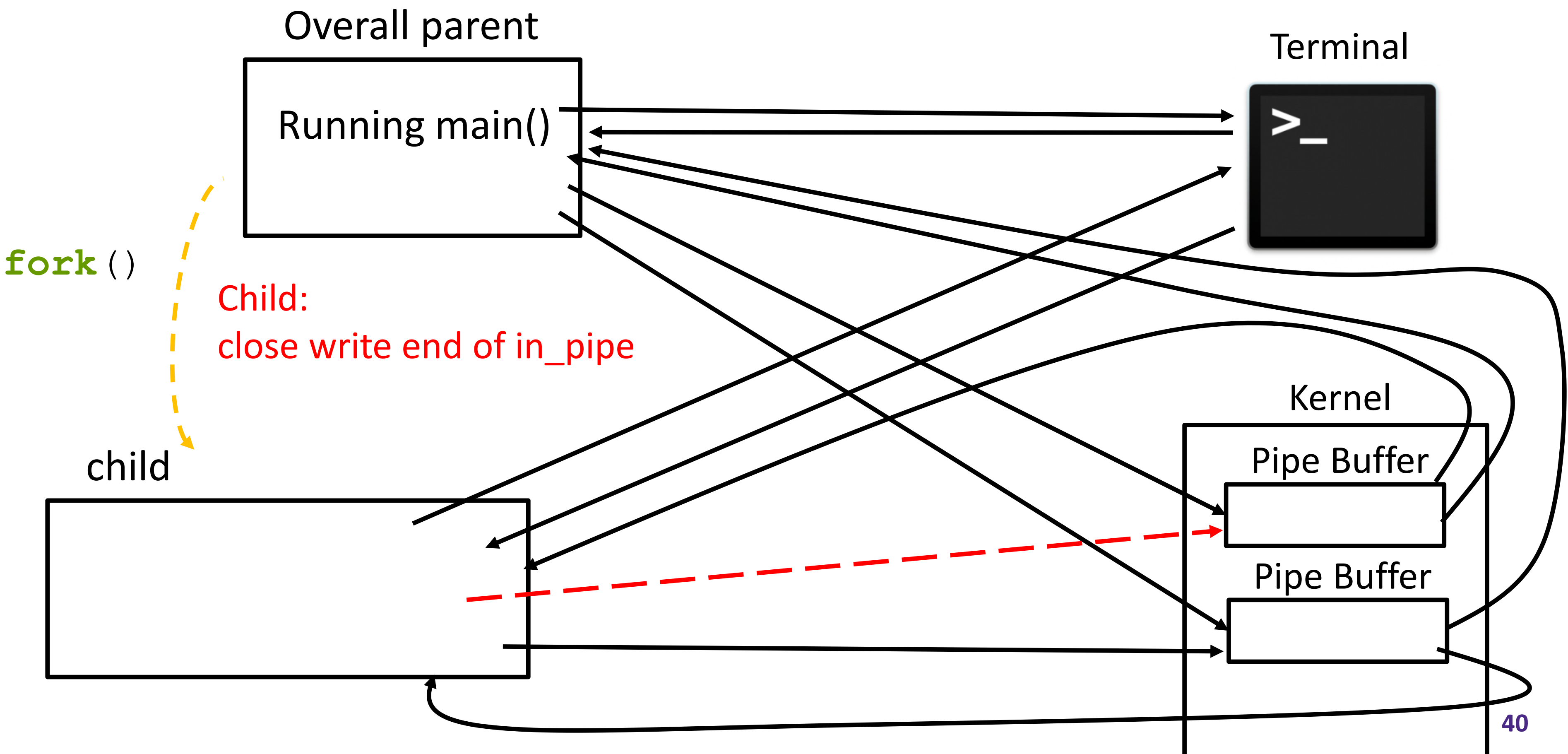
io_autograder.cpp Trace

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



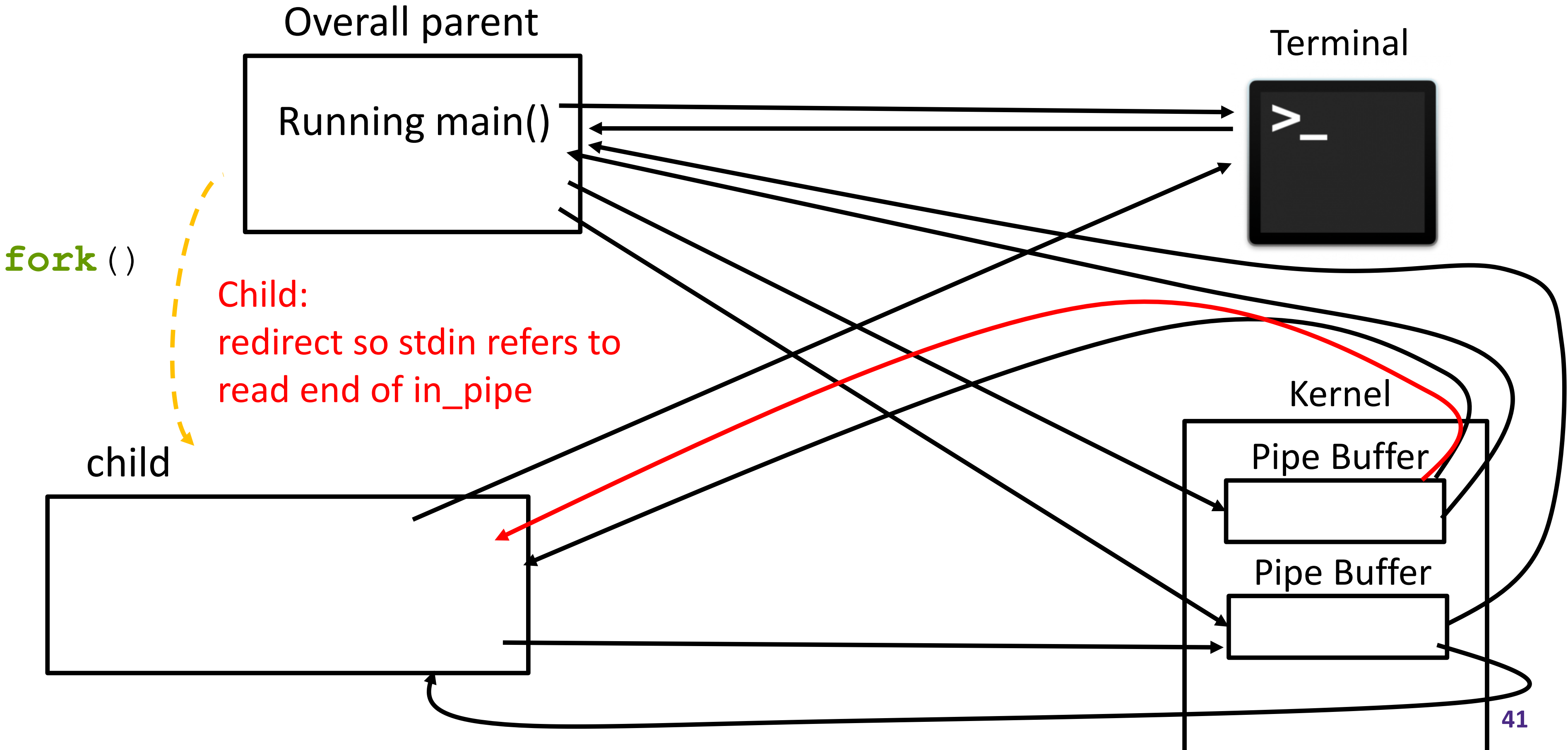
`io_autograder.cpp` Trace

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



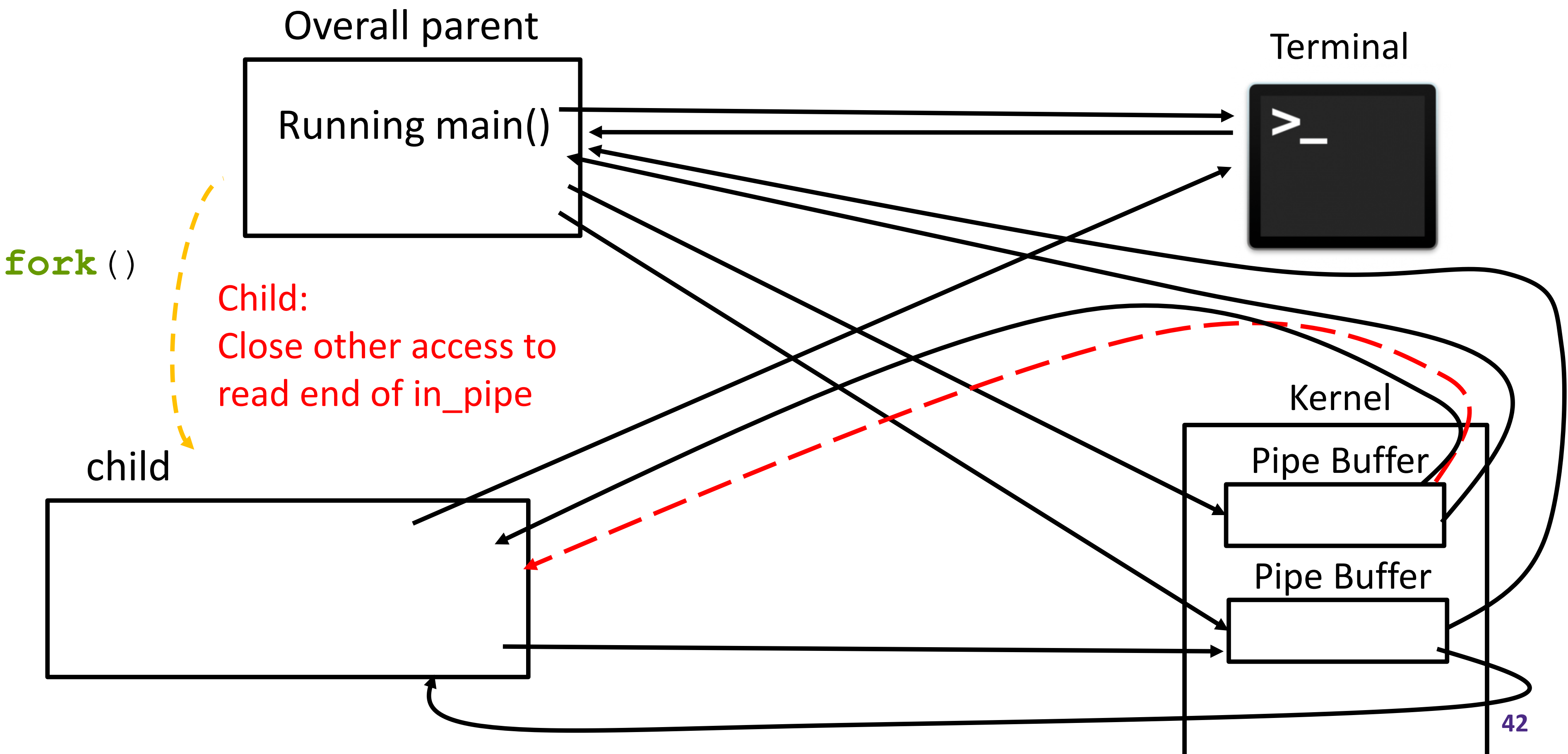
`io_autograder.cpp` Trace

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



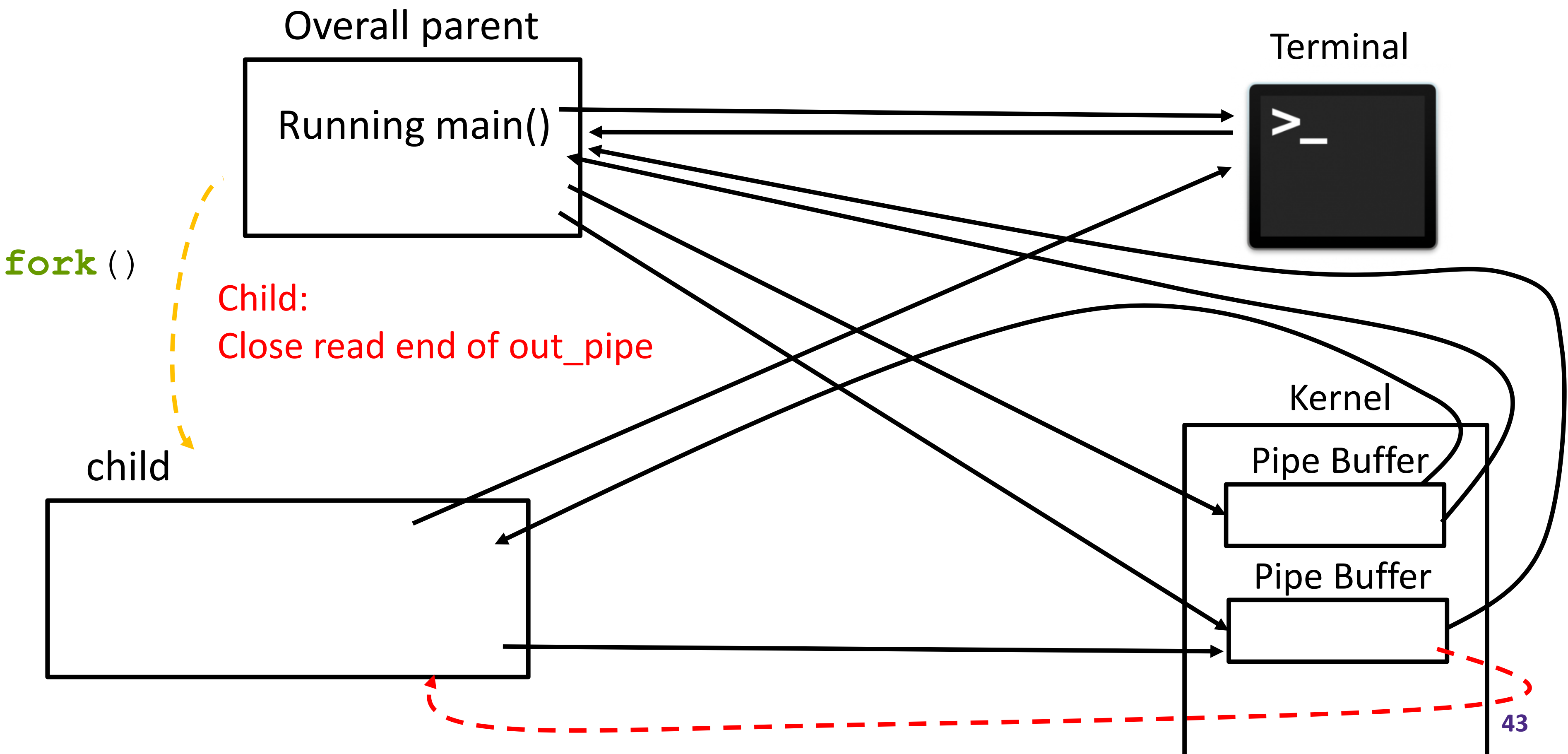
io_autograder.cpp Trace

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



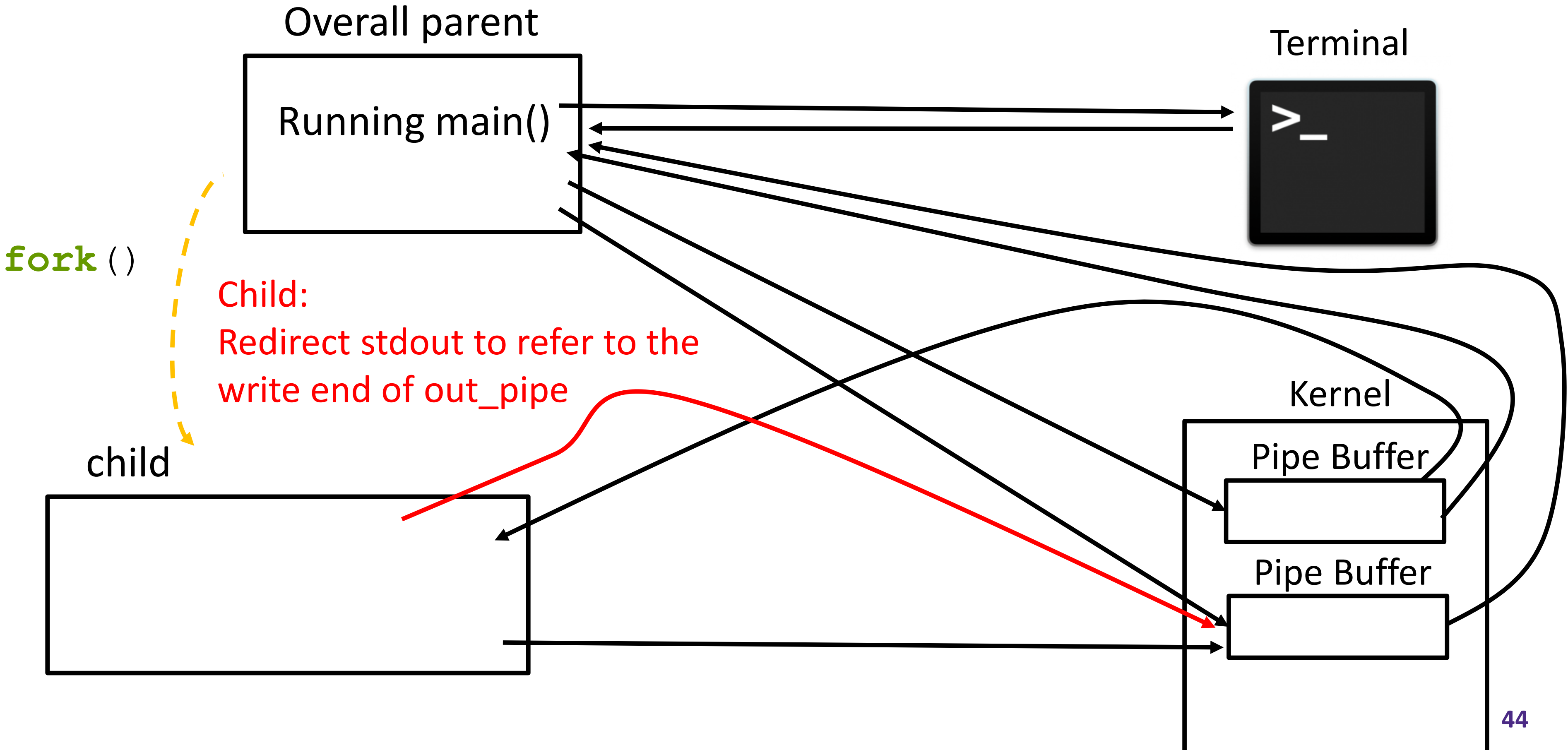
io_autograder.cpp Trace

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



io_autograder.cpp Trace

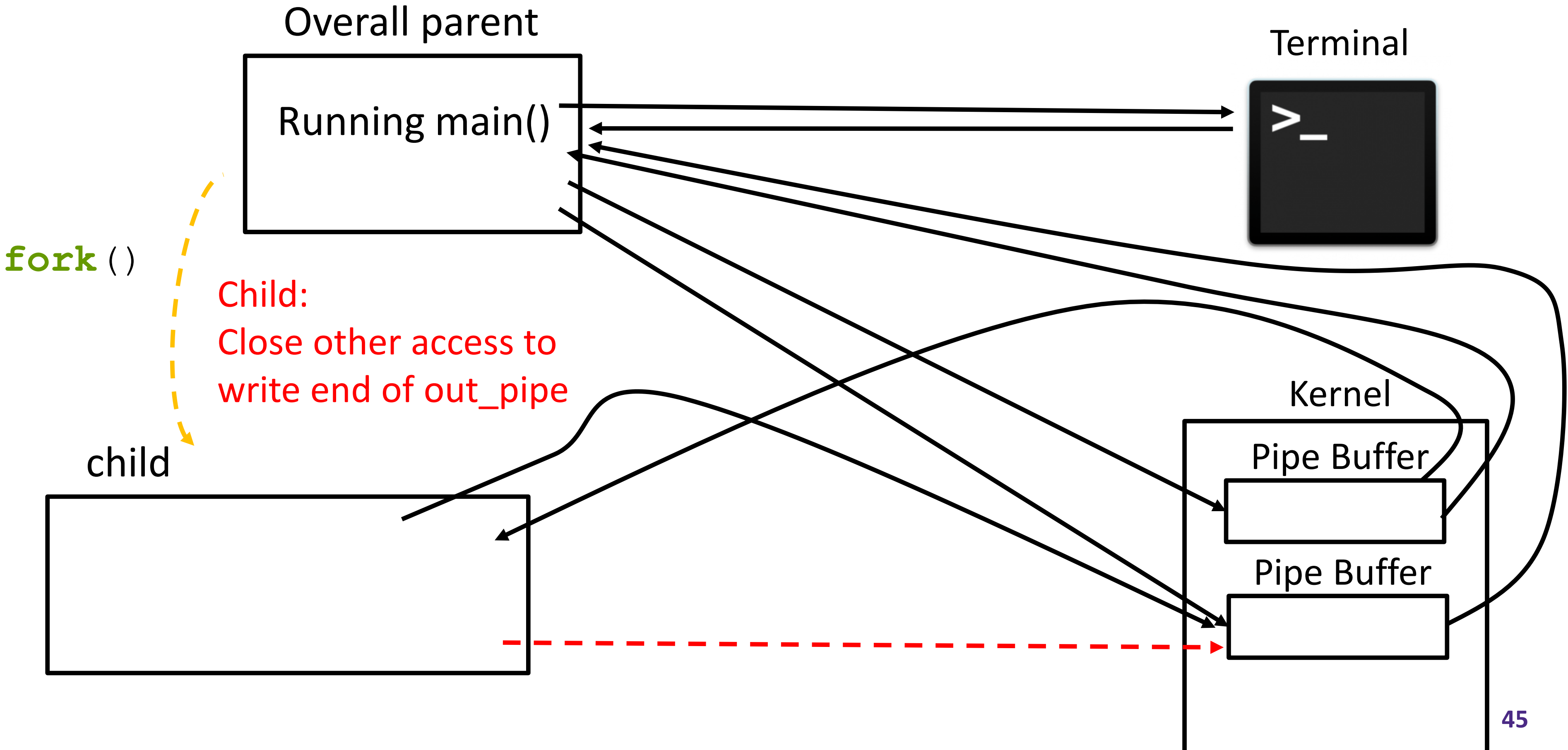
- Compilation done! Run the compiled program… BUT send autograder input and capture output



Child:
Redirect stdout to refer to the write end of out_pipe

`io_autograder.cpp` Trace

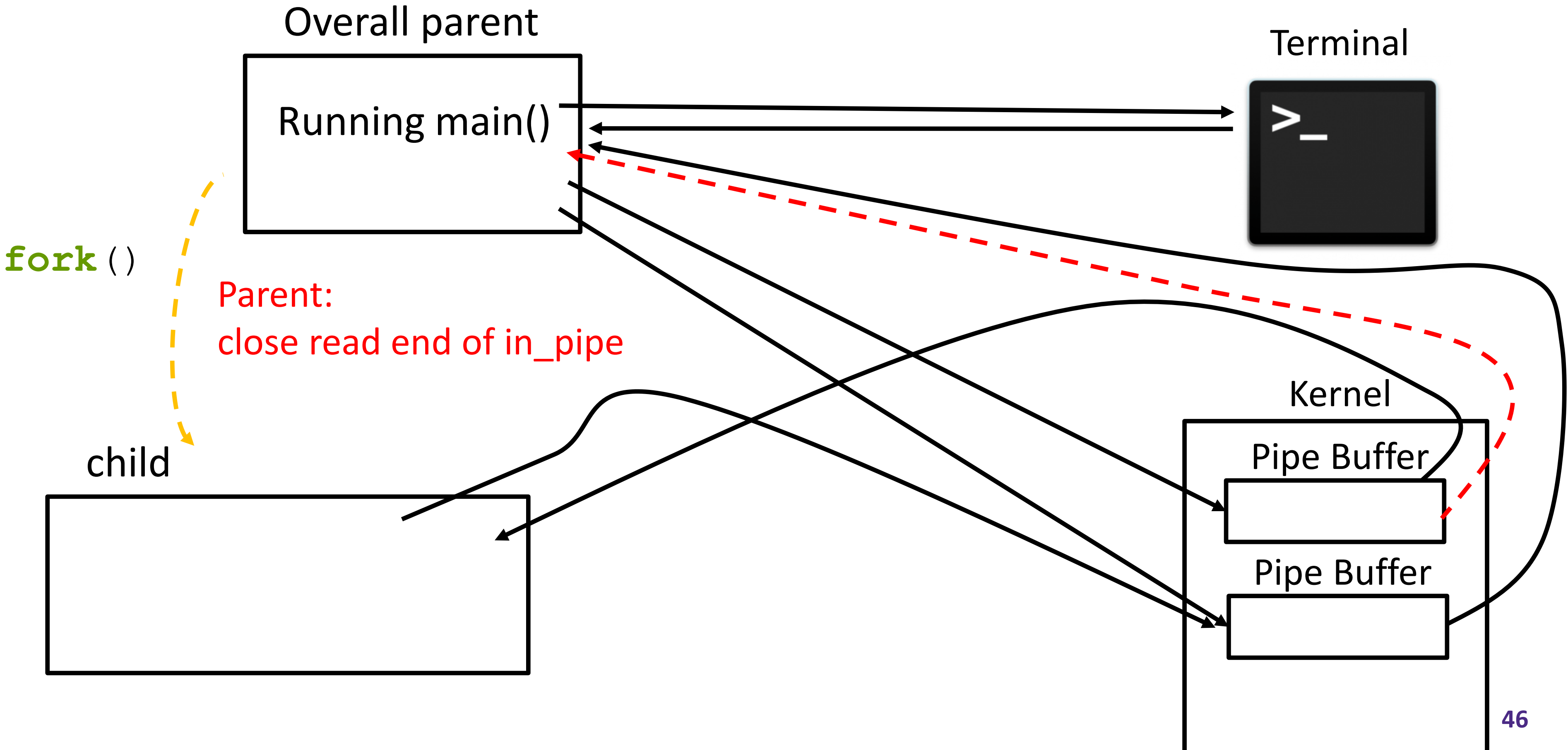
- Compilation done! Run the compiled program… BUT send autograder input and capture output



Child:
Close other access to write end of out_pipe

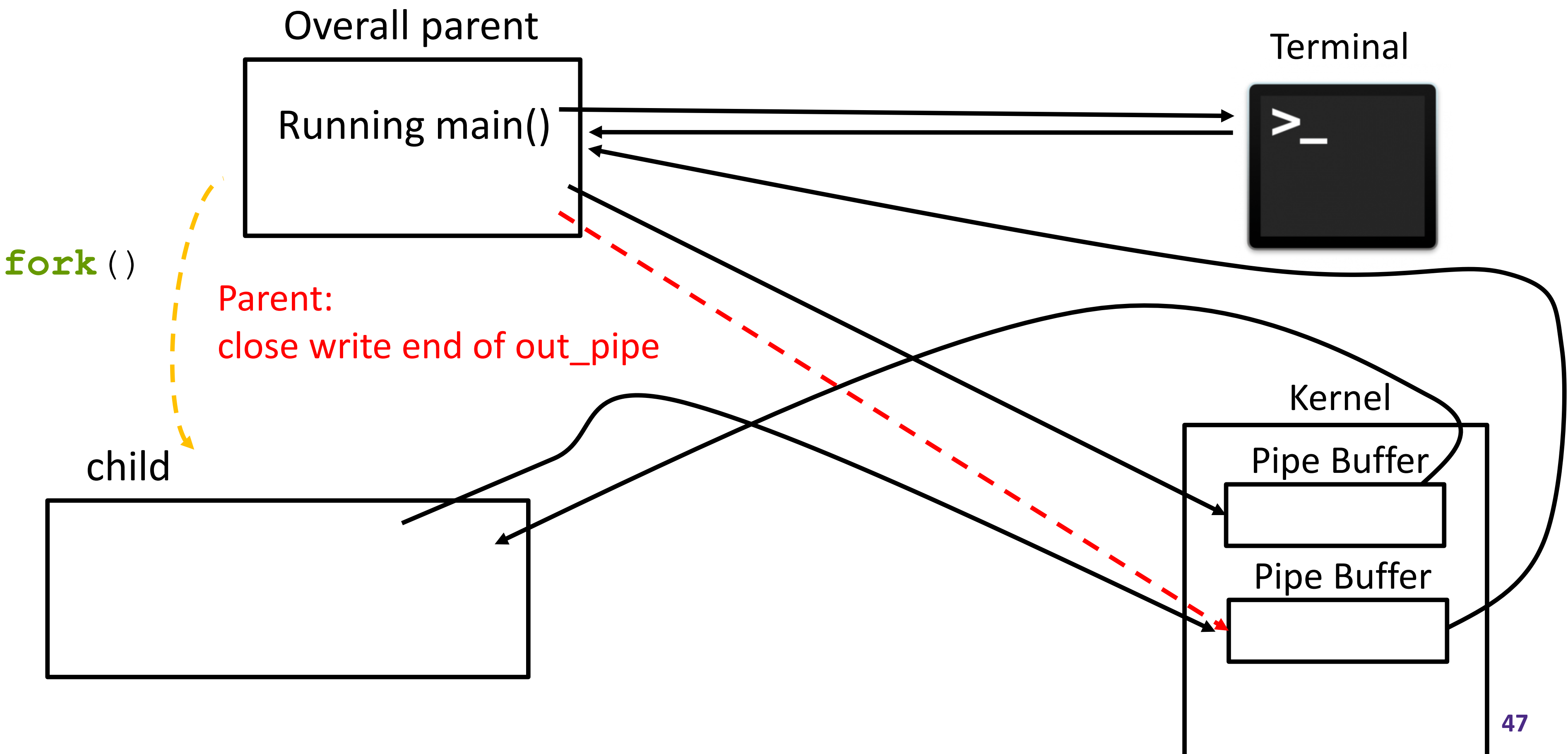
`io_autograder.cpp` Trace

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



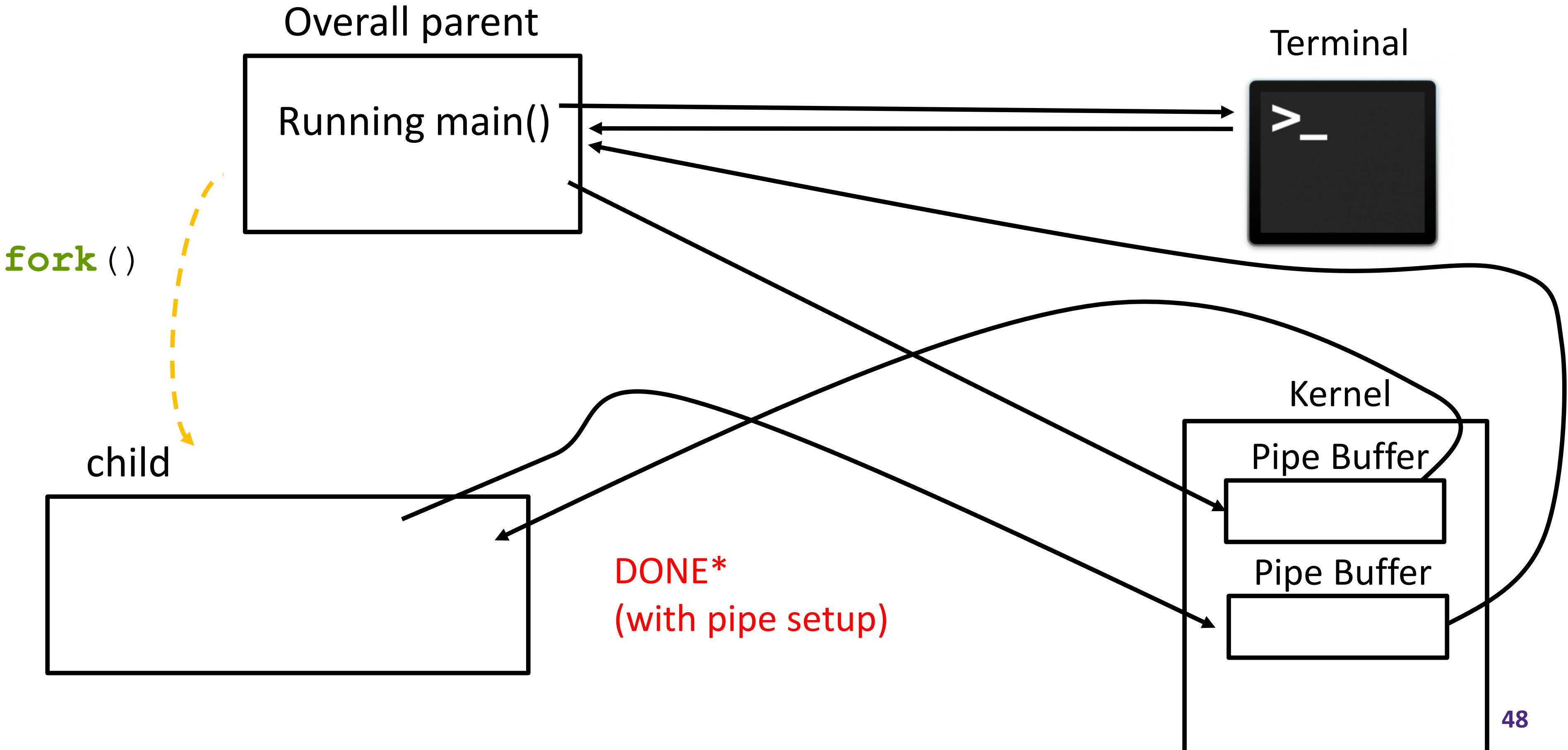
io_autograder.cpp Trace

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



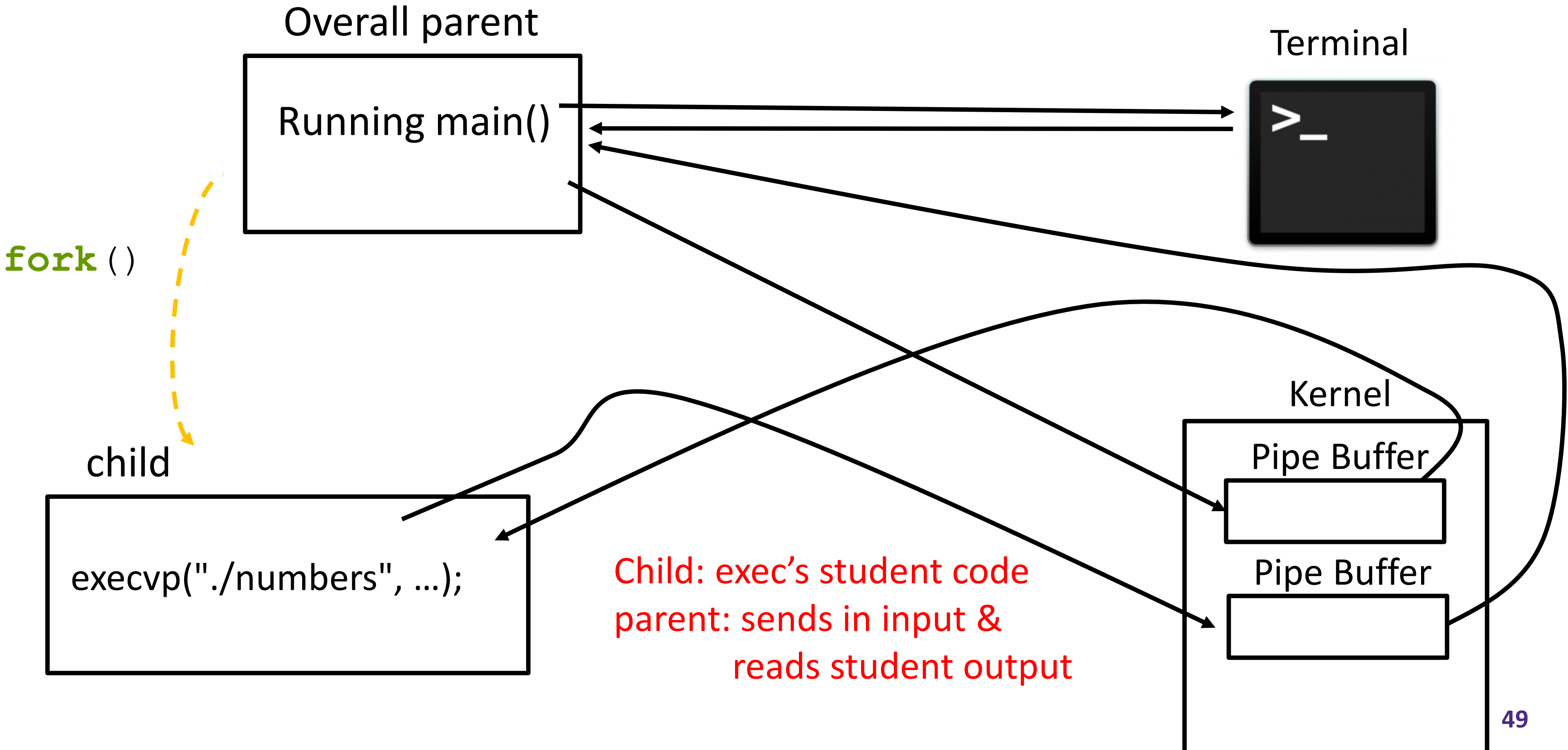
io_autograder.cpp Trace

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



io_autograder.cpp Trace

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



Child: exec's student code
parent: sends in input &
reads student output

Lecture Outline

- Pipe
- **Unix Shell**
- HW4

Unix Shell

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

. / ..

- "/" is used to connect directory and file names together to create a file path.
  - E.g. "**`workspace/595/hello/`**"

- "." is used to specify the current directory.
  - E.g. "**`./test_suite`**" tells to look in the current directory for a file called "**`test_suite`**"

- ".." is like "." but refers to the parent directory.
  - E.g. "**`./solution_binaries/../test_suite`**" would be effectively the same as the previous example.

Common Commands (Pt. 1)

- "**`ls`**" lists out the entries in the specified directory (or current directory if another directory is not specified

- "**`cd`**" changes directory to the specified directory
  - E.g. "**`cd ./solution_binaries`**"

- "**`exit`**" closes the terminal

- "**`mkdir`**" creates a directory of specified name

- "**`touch`**" creates a specified file. If the file already exists, it just updates the file's time stamp

Common Commands (Pt. 2)

- "**`echo`**" takes in command line args and simply prints those args to stdout
  - "**`echo hello!`**" simply prints "**`hello!`**"

- "**`wc`**" reads a file or from stdin some contents. Prints out the line count, word count, and byte count

- "**`cat`**" prints out the contents of a specified file to stdout. If no file is specified, prints out what is read from stdin

- "**`head`**" print the first 10 line of specified file or stdin to stdout

Common Commands (Pt. 3)

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

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

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

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

Unix Shell Commands

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

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

- 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-shredder < test_case`"

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

- Complex example:

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

Poll Everywhere

pollev.com/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

- 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]

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

D. ls && wc

[✓ E. The correct answer is not listed]

F. We're lost…

`ls | wc -l`
would be preferred.

Lecture Outline

- Pipe
- Unix Shell
- **HW4**

HW4 Demo

- In HW4, 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.

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`"

HW4 Demo

- In HW4, 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.

- Can run a sample solution with:

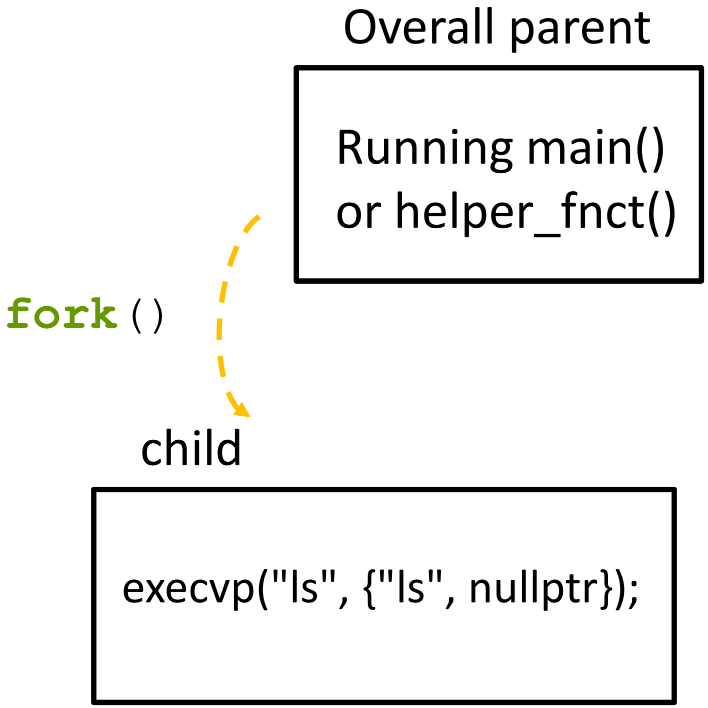
```
./solution_binaries/pipe_shell
```

Suggested Approach

- HIGHLY ENCOURAGED to follow the suggested approach
  - Write a program that acts similarly to stdin_echo.cc
  - 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"`

HW4 Example Line

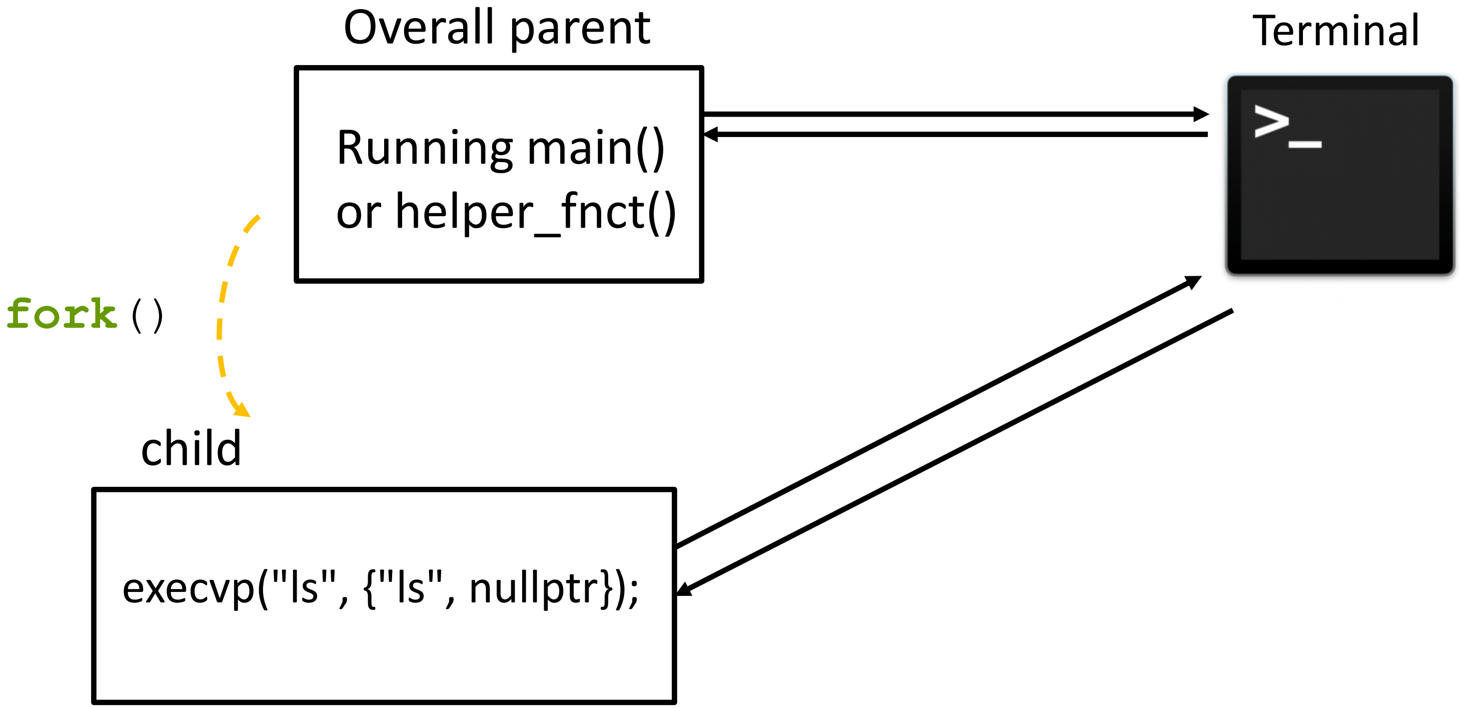
- Consider the case when a user inputs
  - `"ls"`



```
execvp("ls", {"ls", nullptr});
```

HW4 Example Line

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



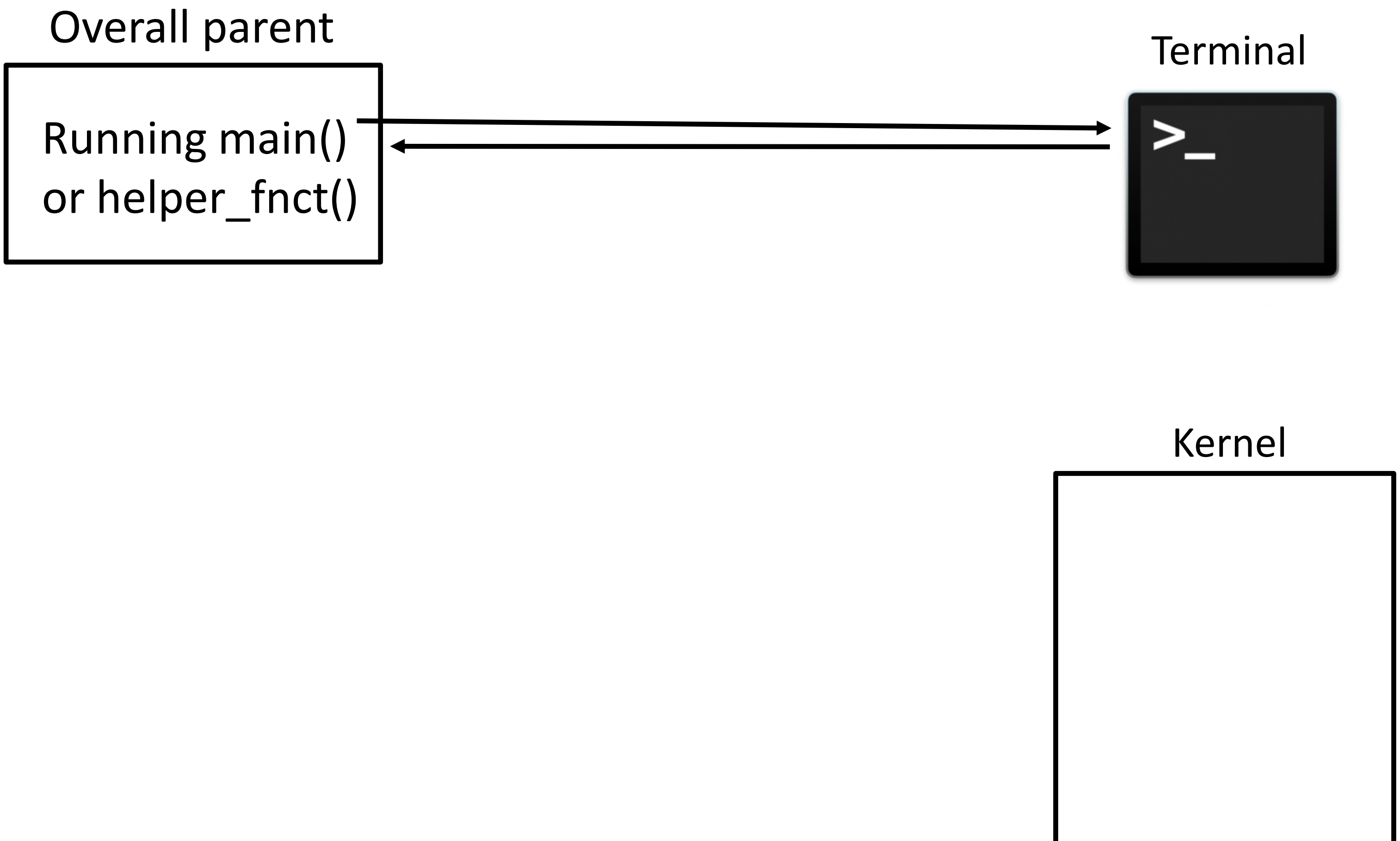
HW4 Hints

- 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

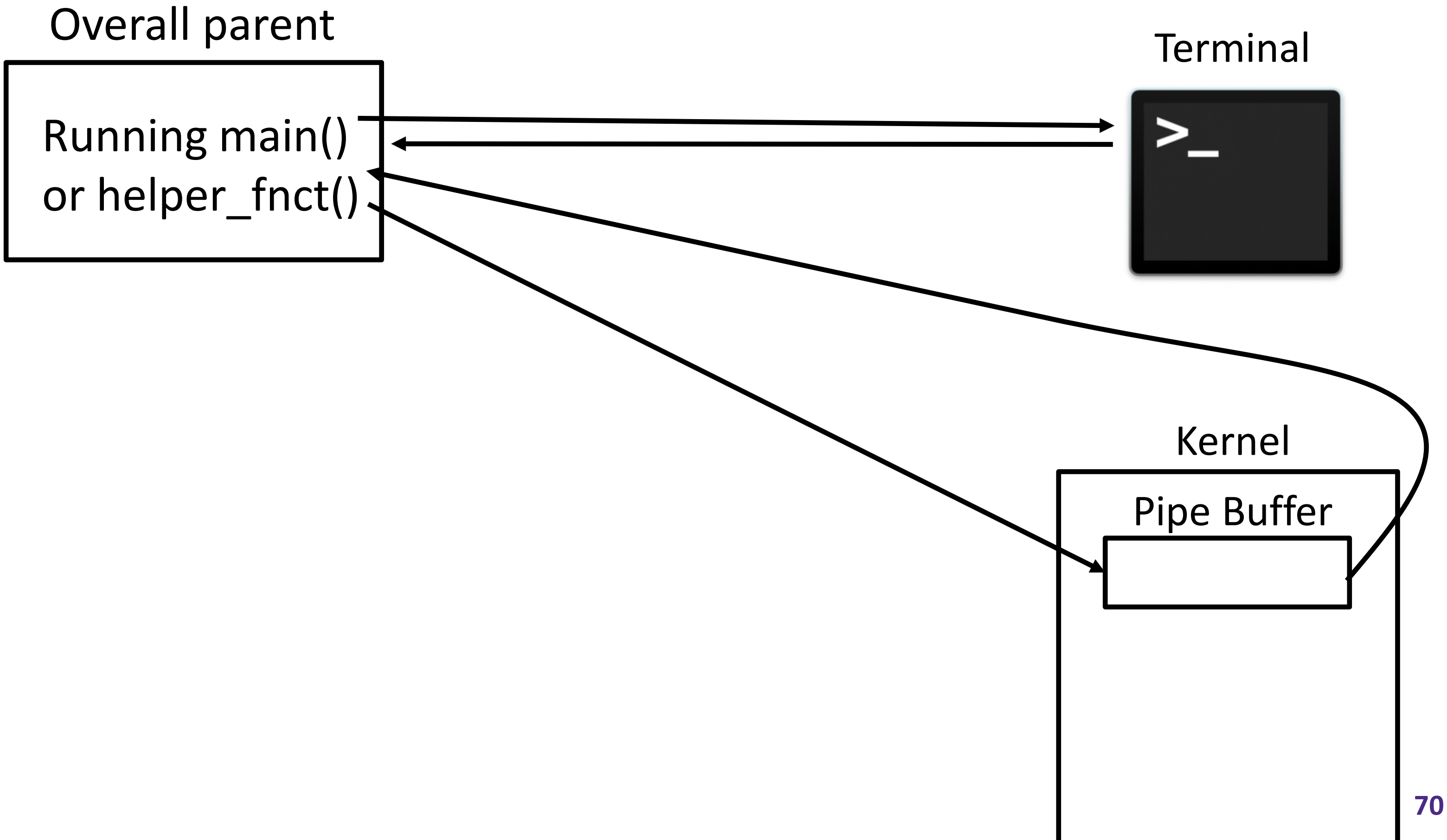
HW4 Example Line 1

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



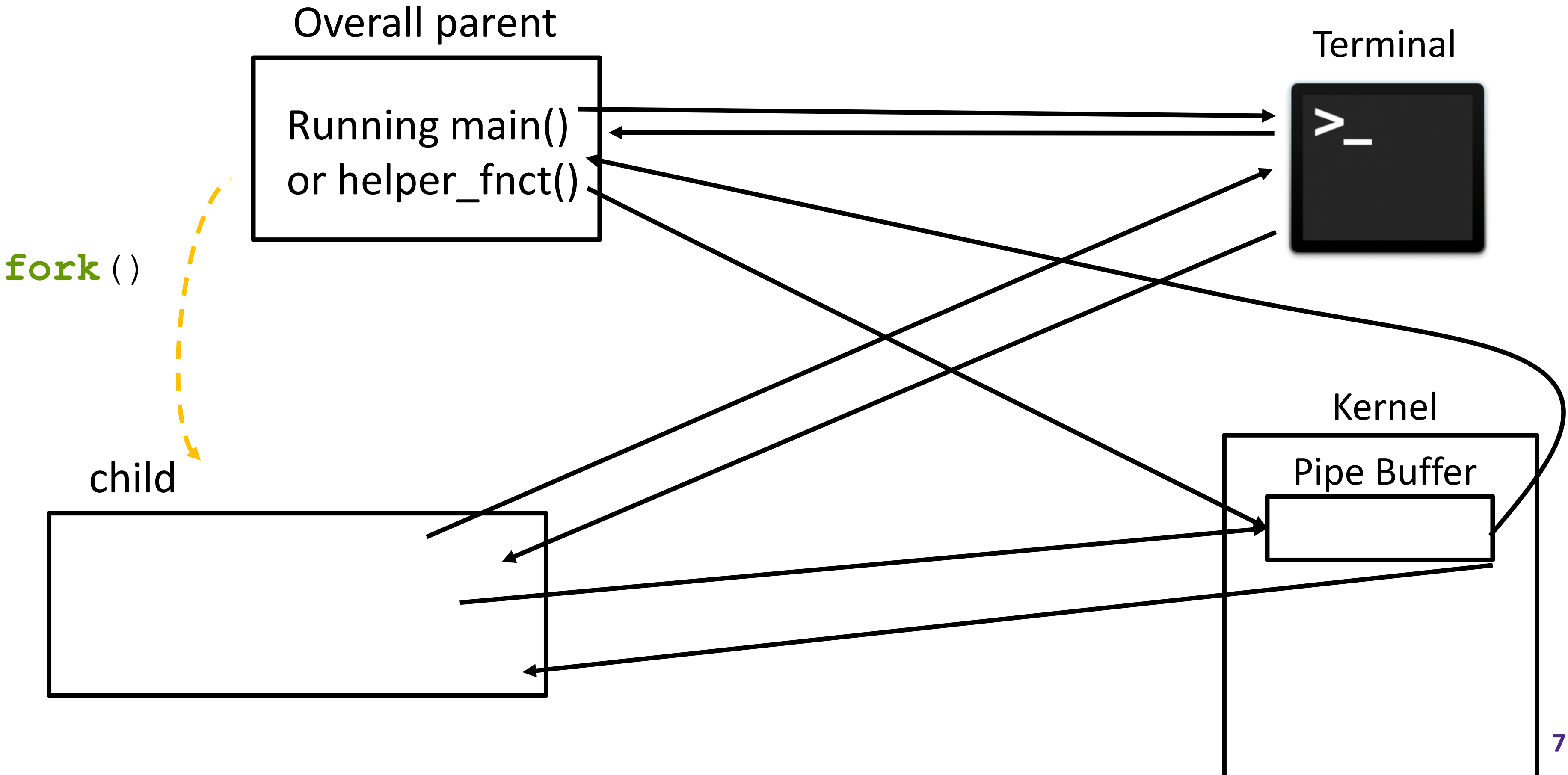
HW4 Example Line 1

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



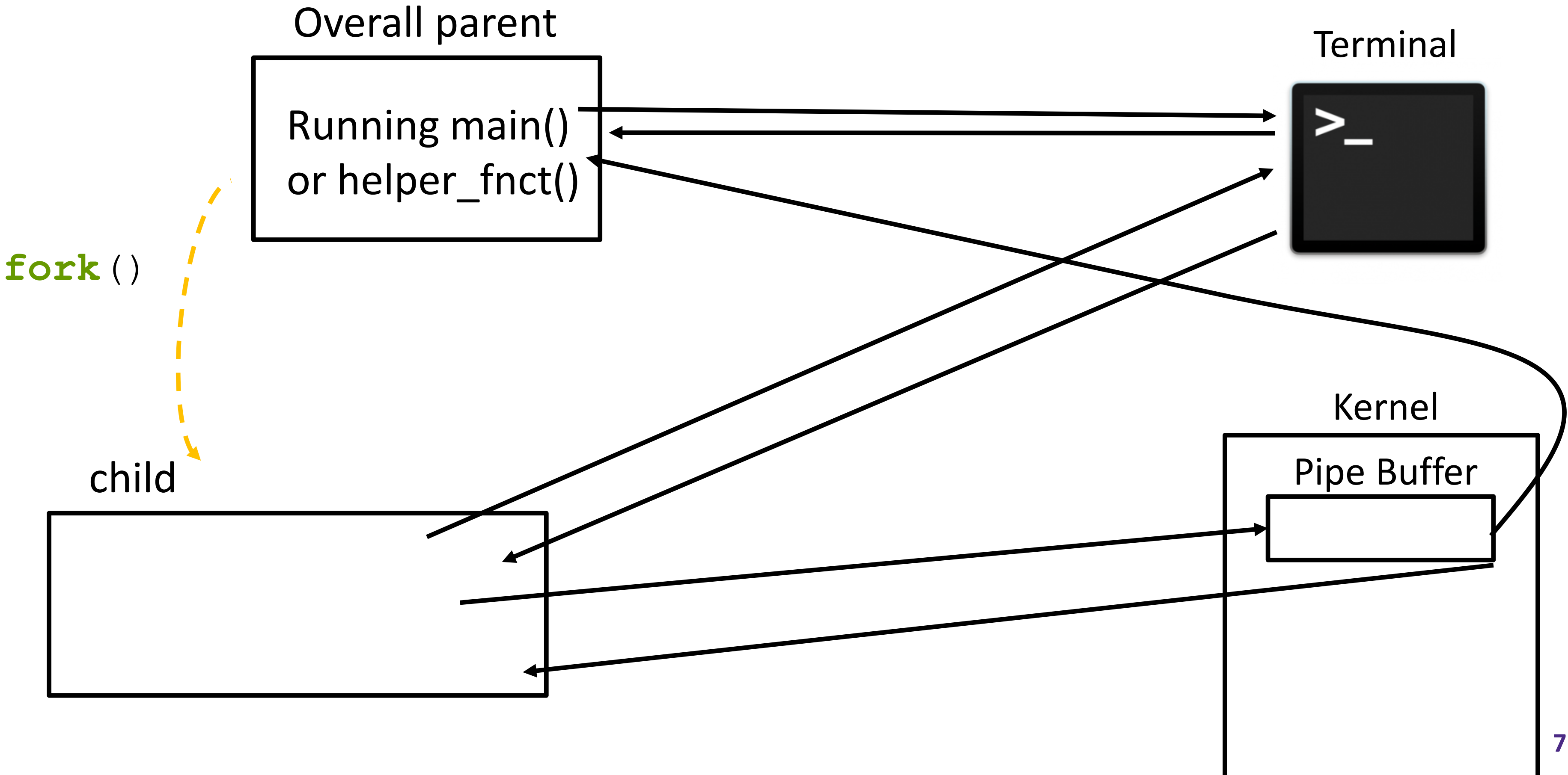
HW4 Example Line 1

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



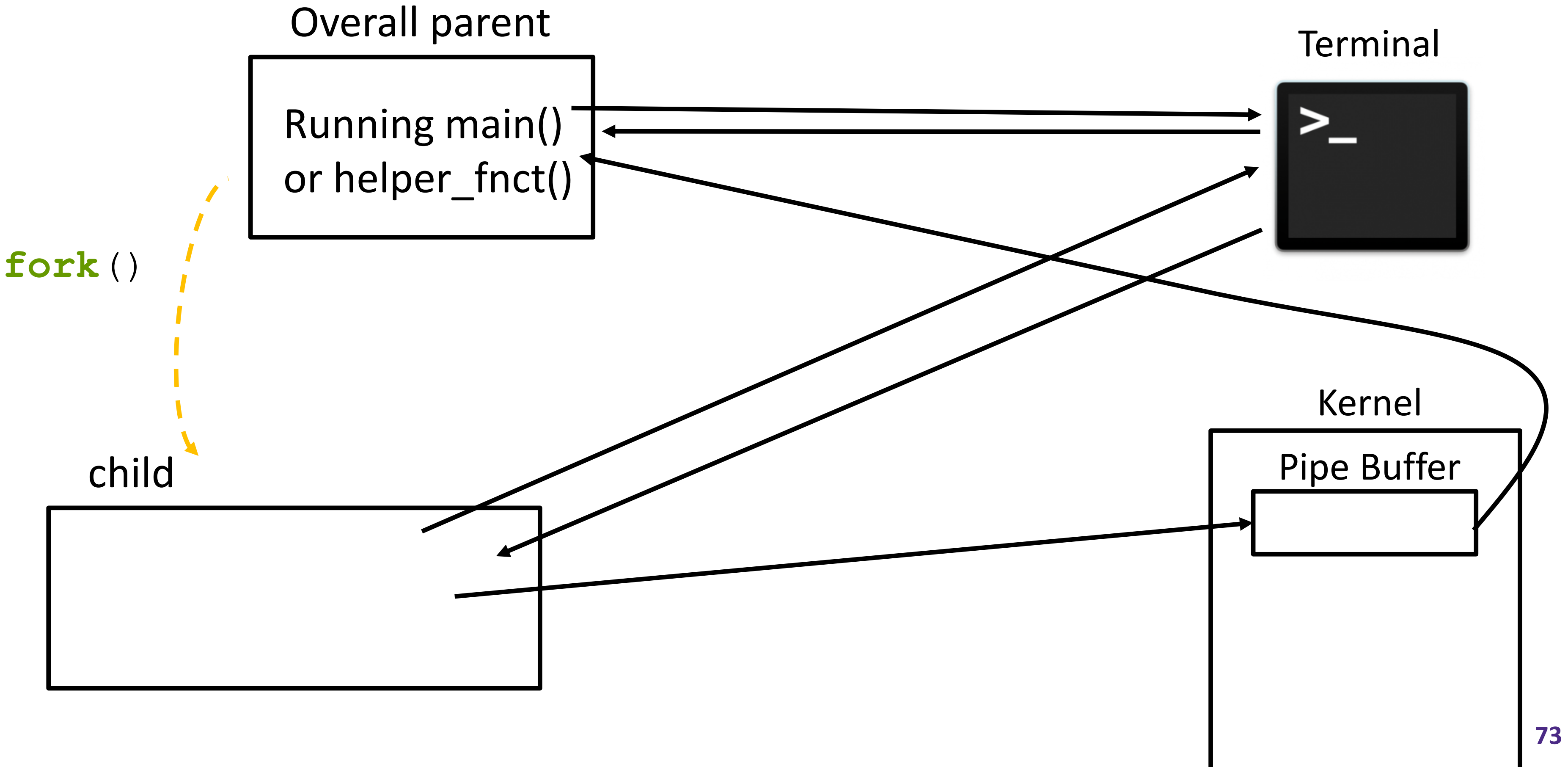
HW4 Example Line 1

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



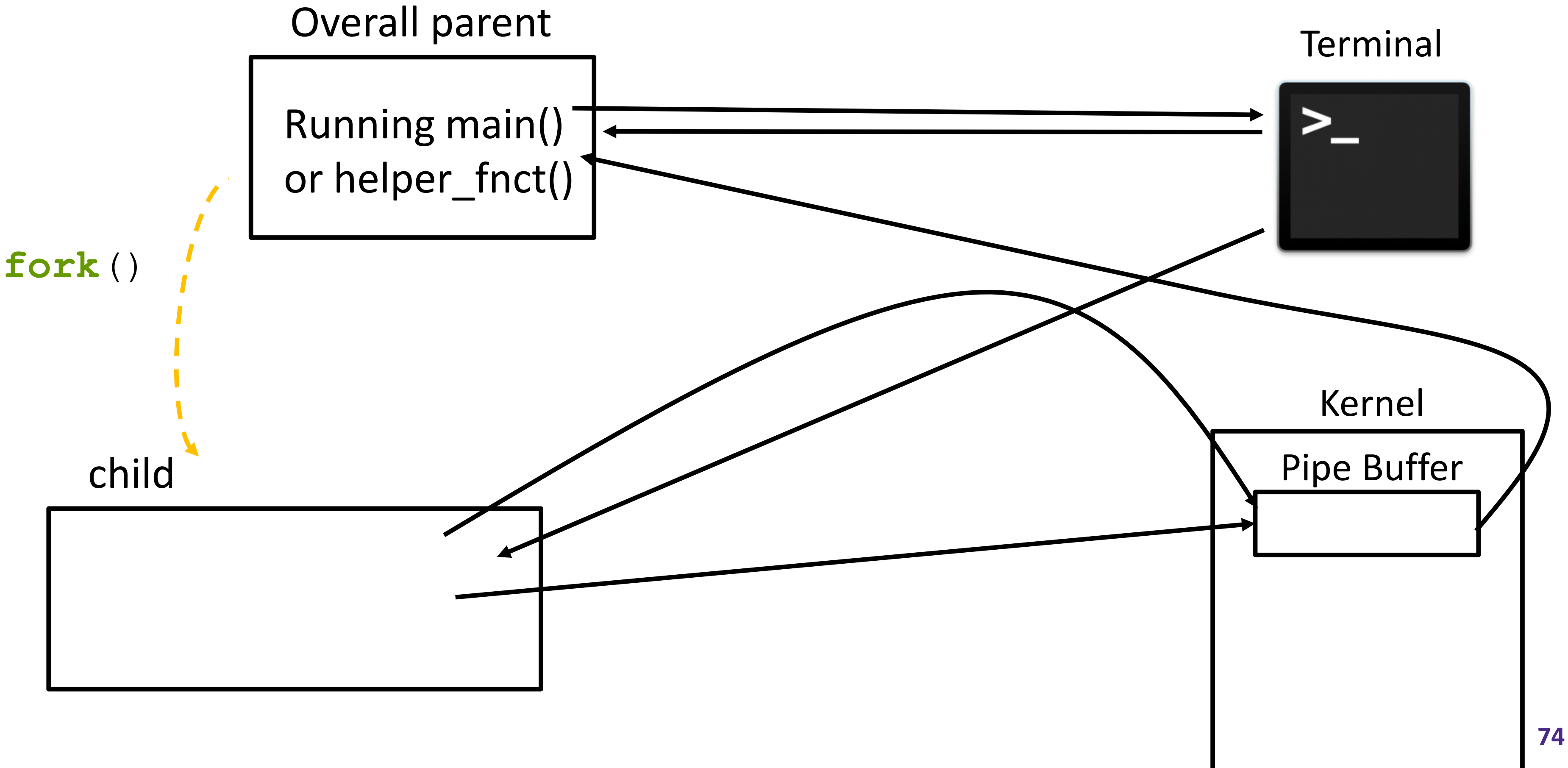
HW4 Example Line 1

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



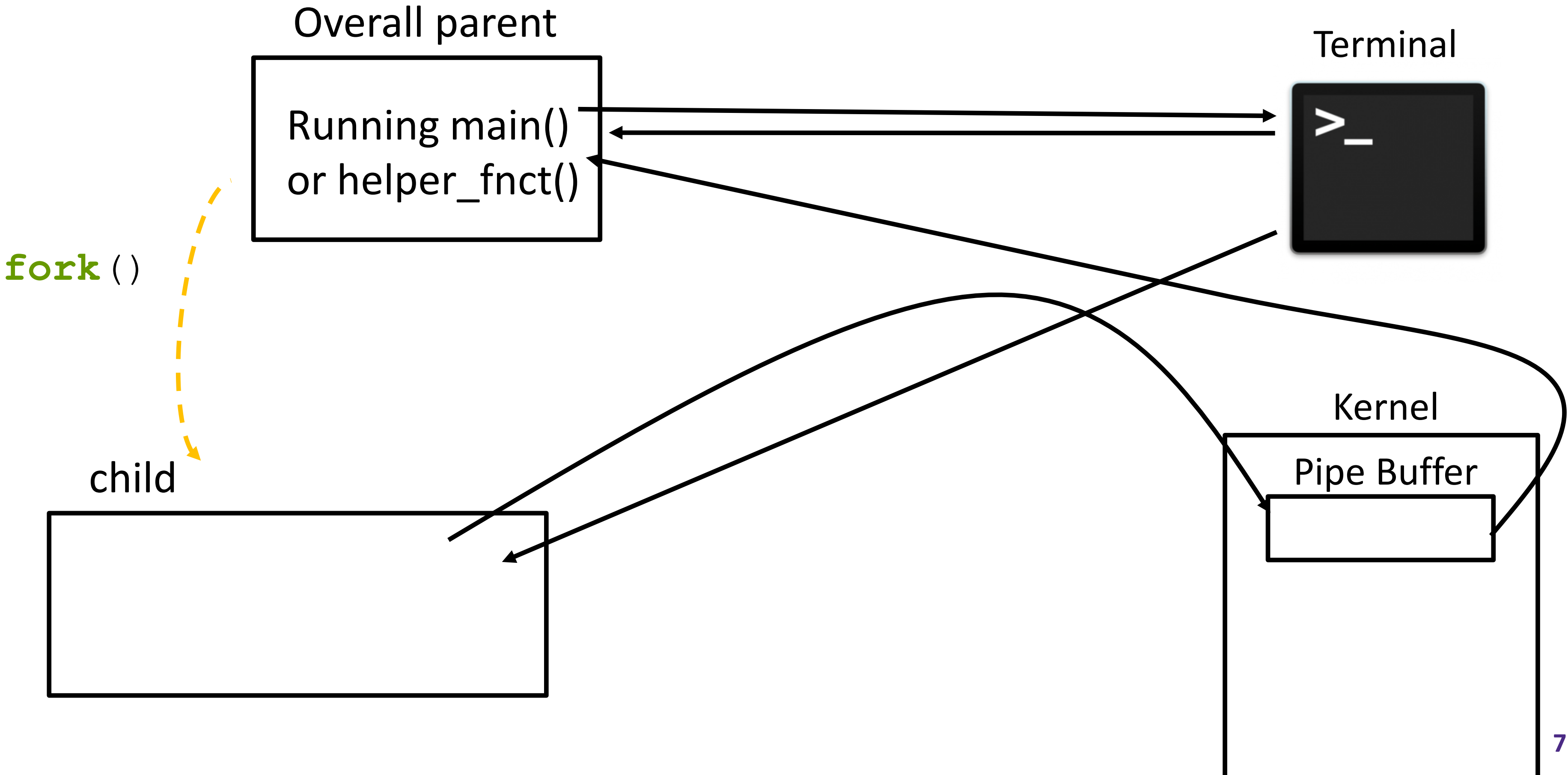
HW4 Example Line 1

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



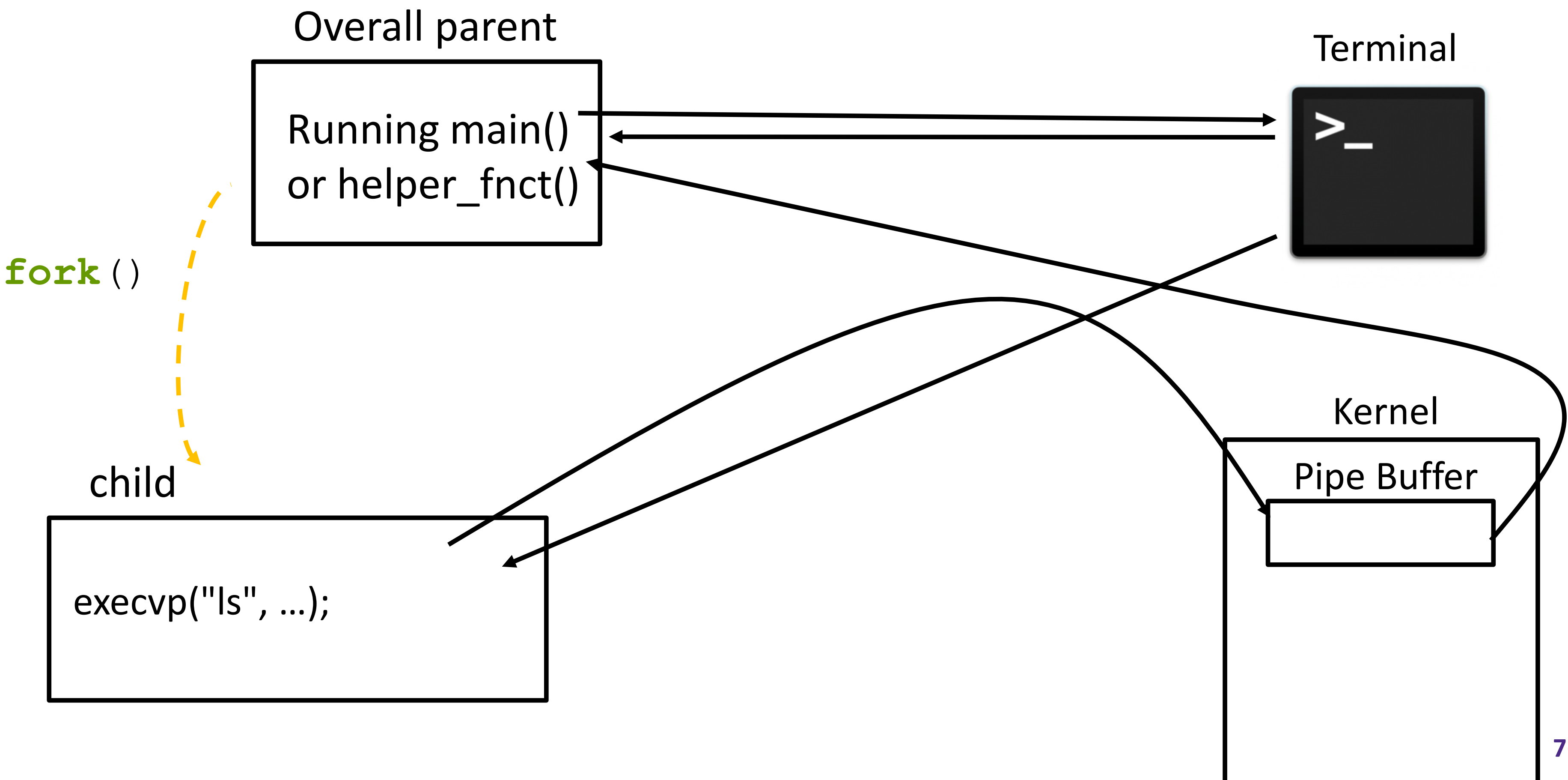
HW4 Example Line 1

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



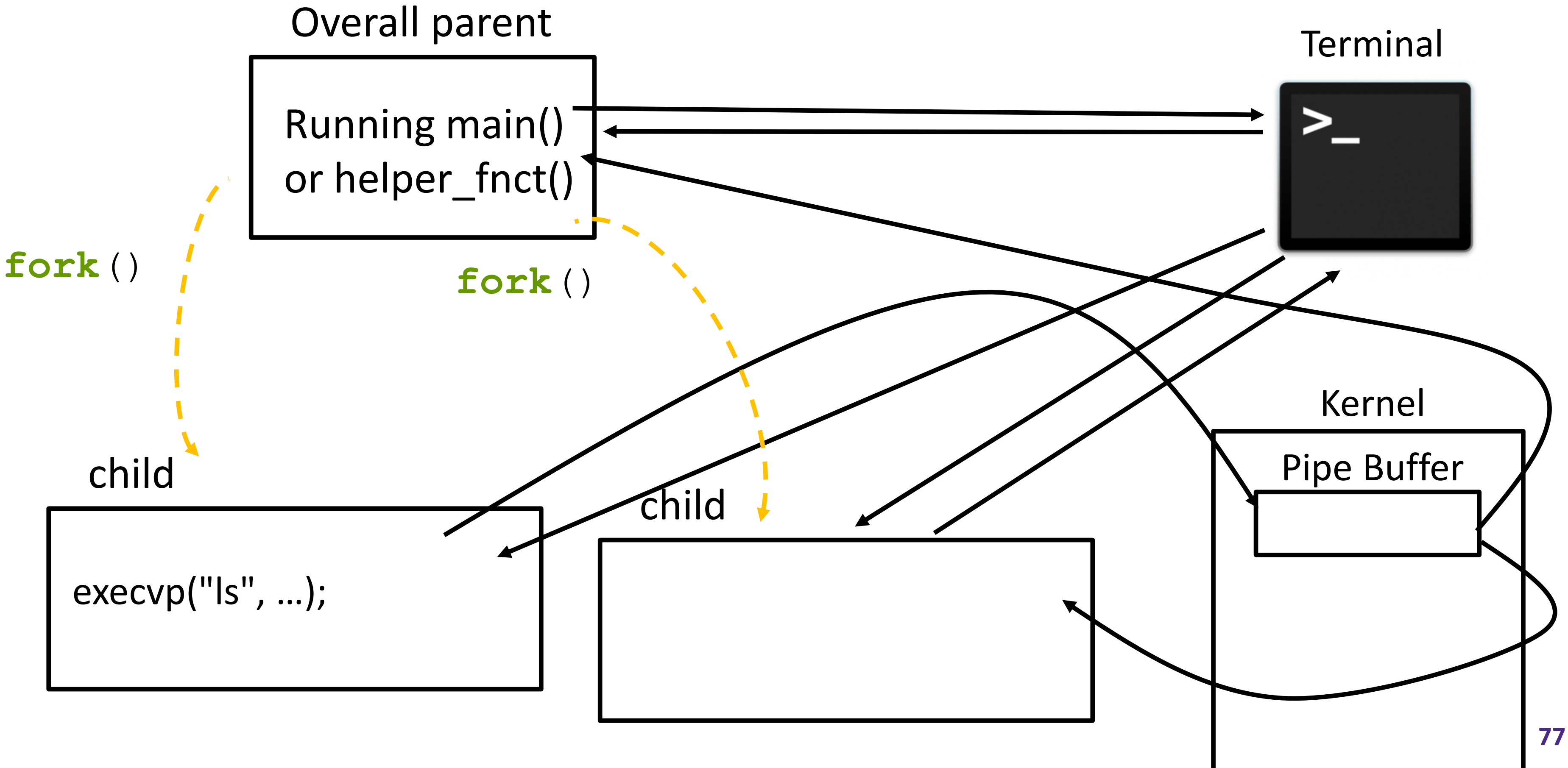
HW4 Example Line 1

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



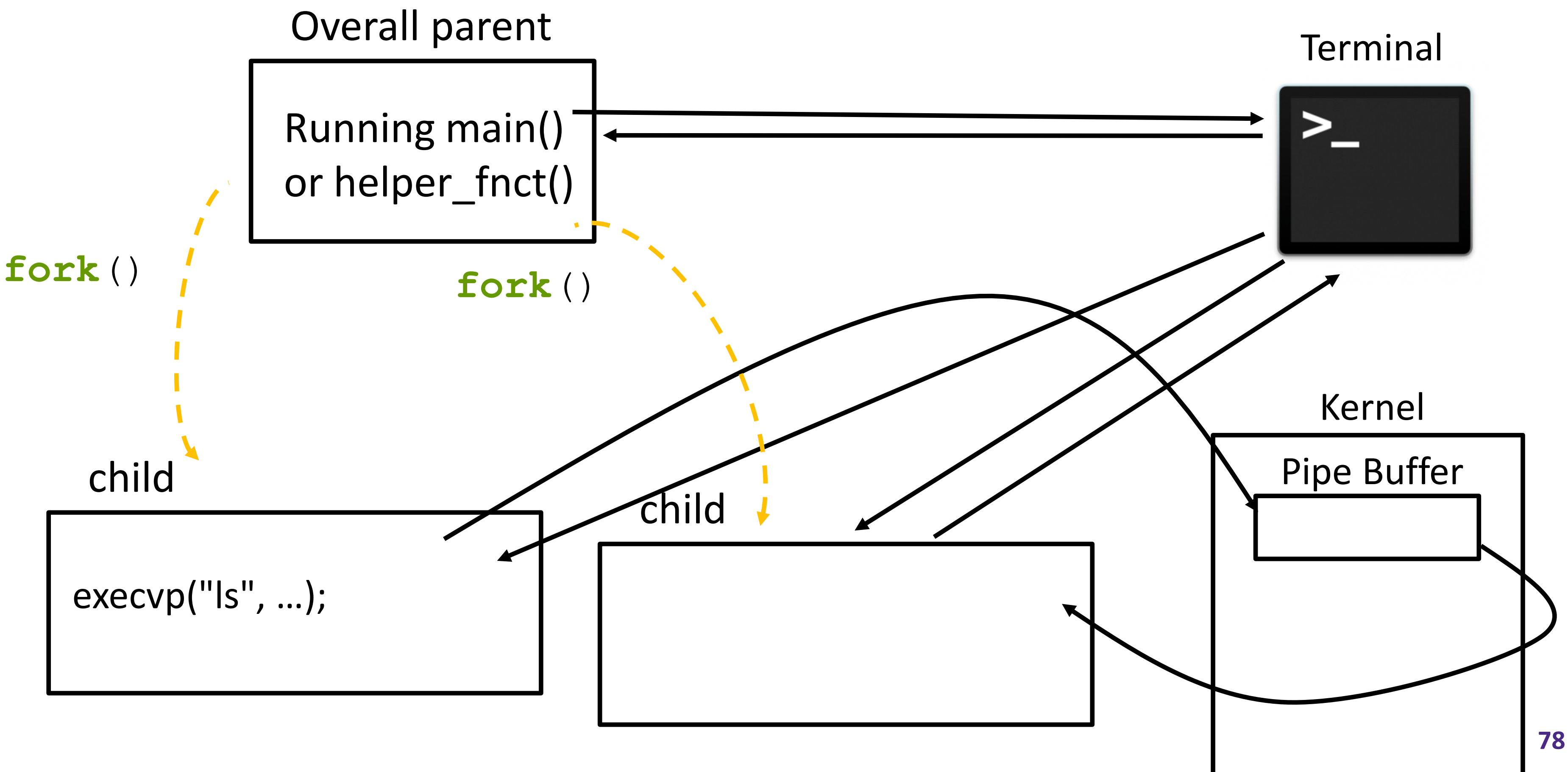
HW4 Example Line 1

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



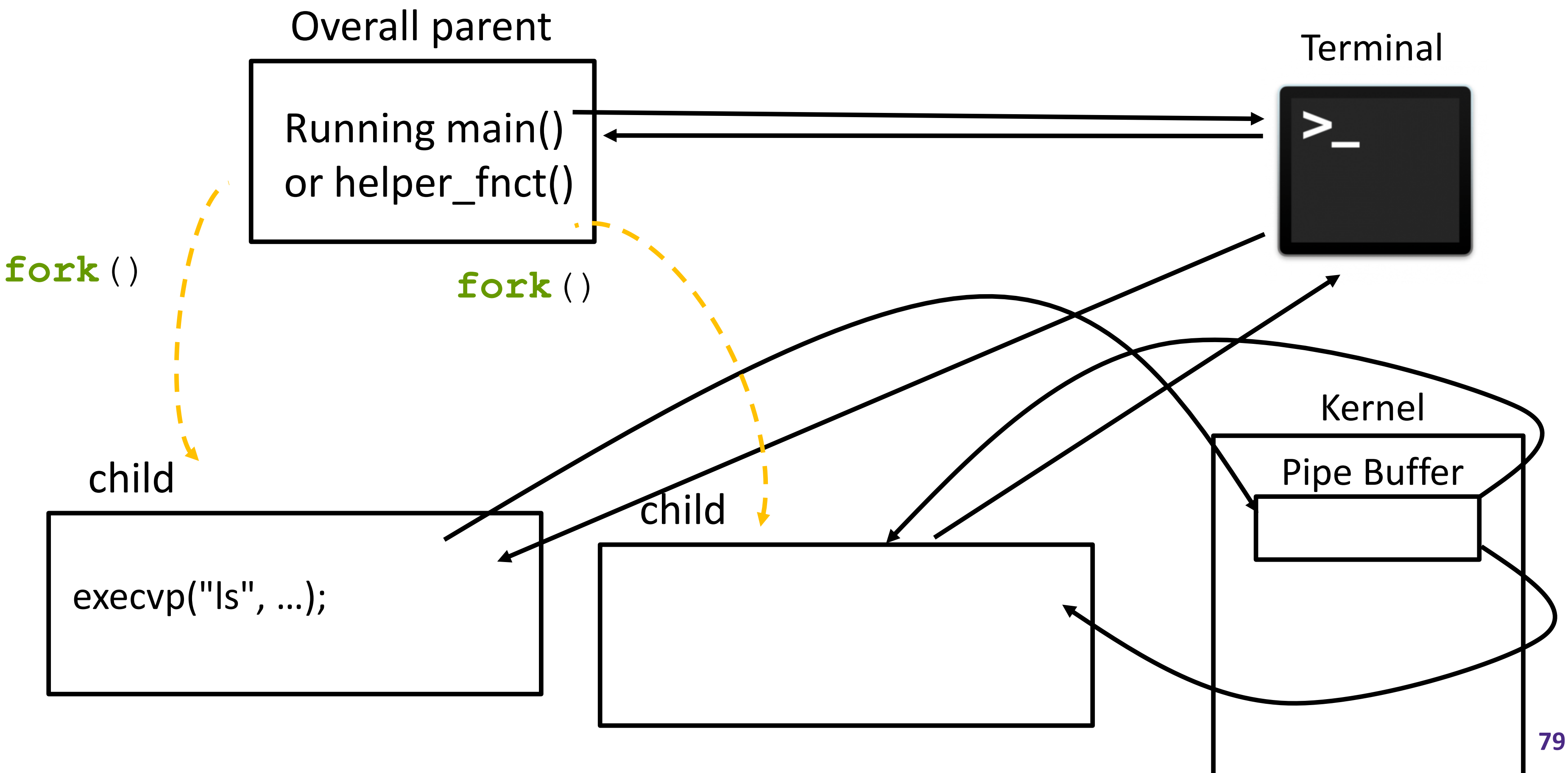
HW4 Example Line 1

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



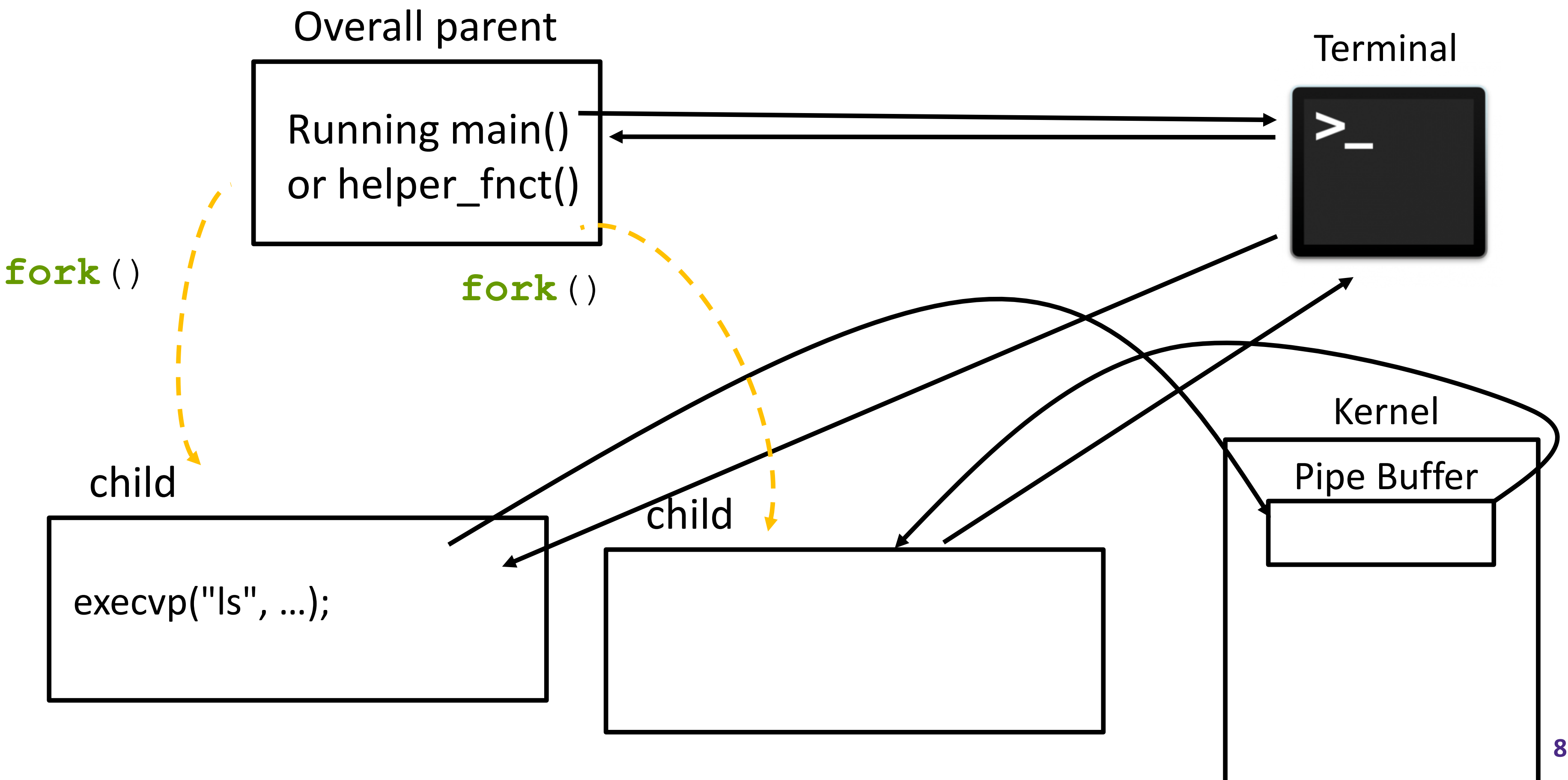
HW4 Example Line 1

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



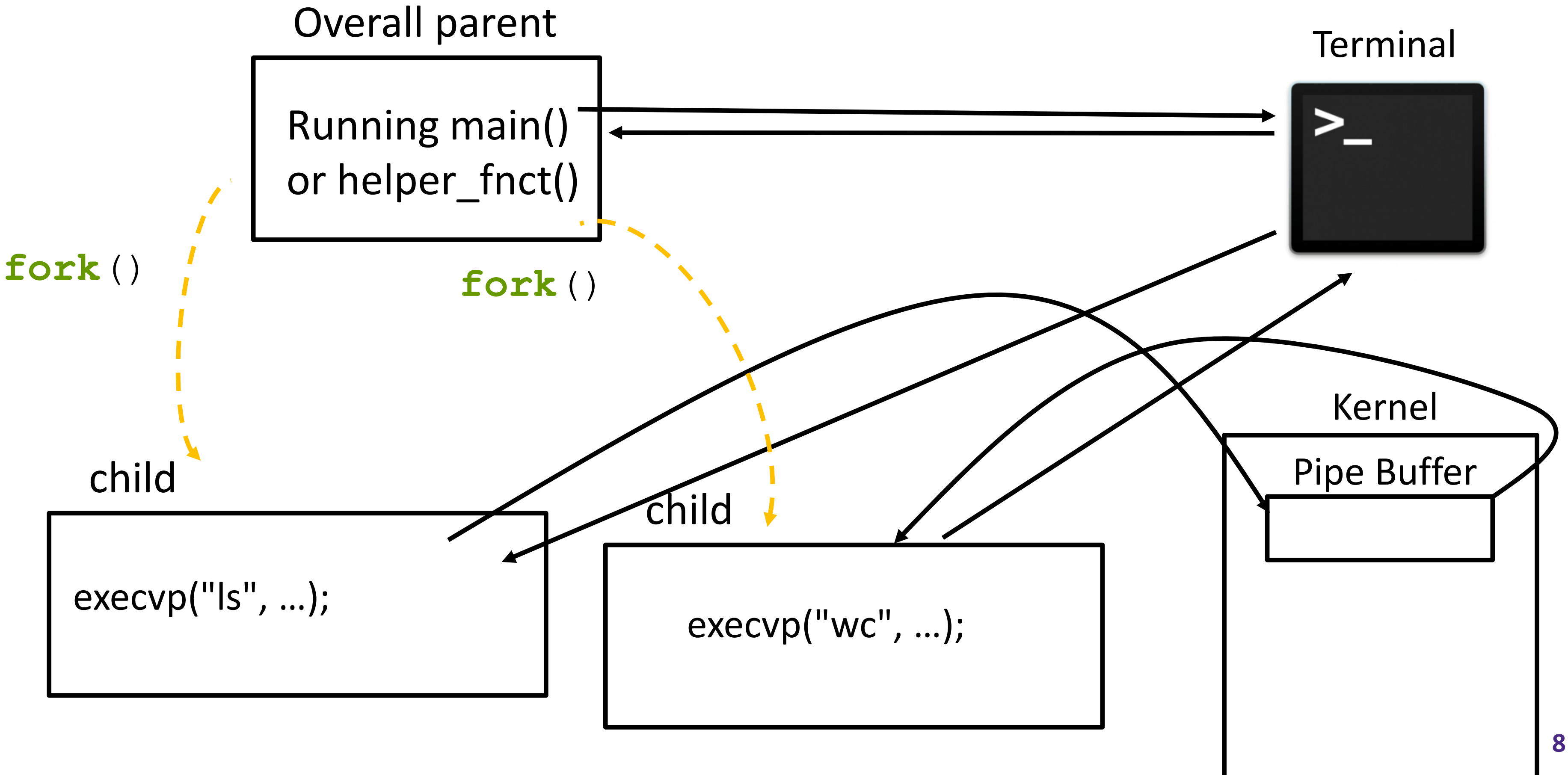
HW4 Example Line 1

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



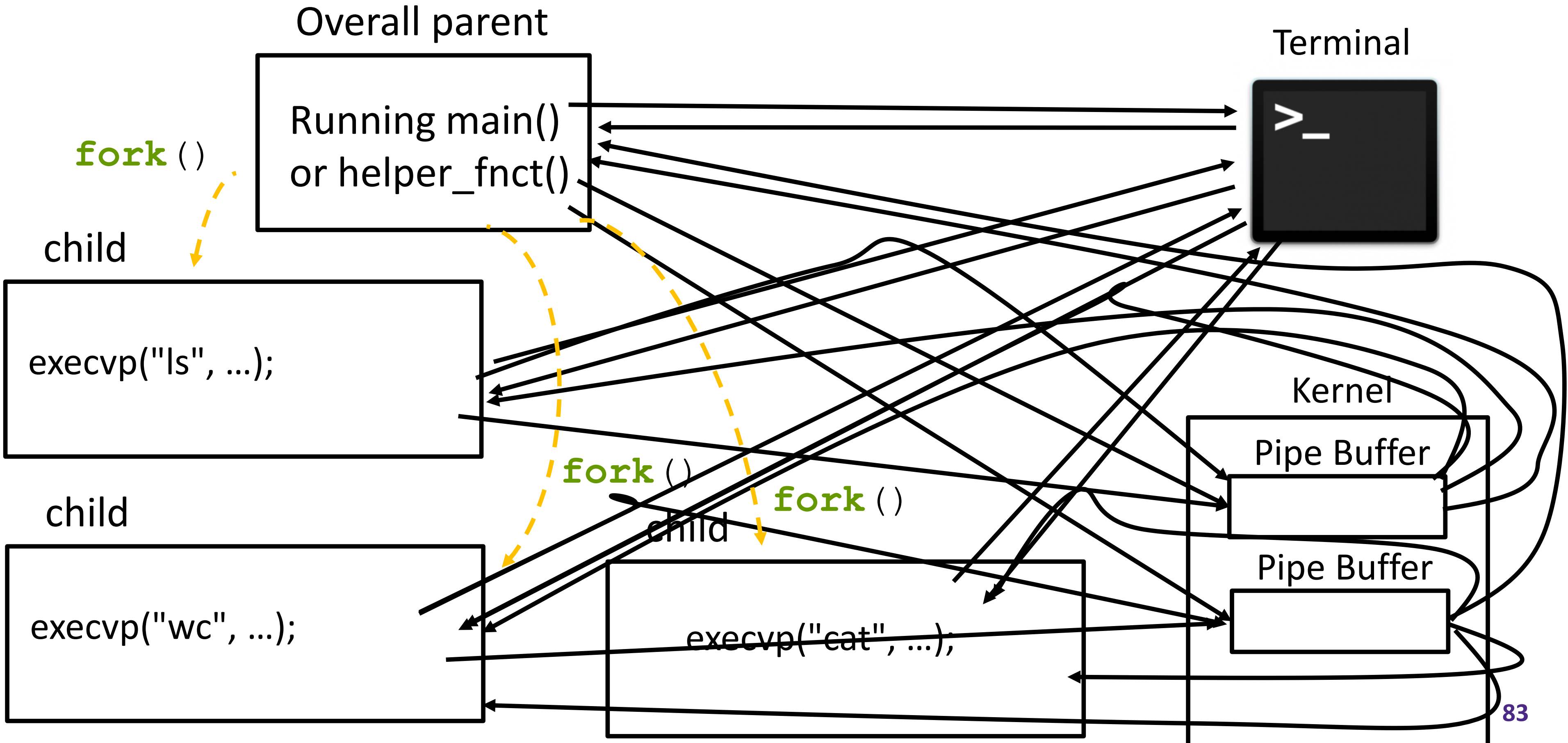
HW4 Example Line 1

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



HW4 Example Line 2

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


Overall parent
Running main()
or helper_fnct()
Terminal
>_
fork()
child
execvp("ls", ...);
child
execvp("wc", ...);
fork()
fork()
child
execvp("cat", ...);
Kernel
Pipe Buffer
Pipe Buffer