Project Overview & pipe()

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

- Exam grades posted Monday night
  - Regrade Requests open till Tuesday 4/9 @ 11:59pm)
  - Rember that we have the clobber policy, it is ok if the exam did not go well.
- HW03 due Friday this week
  - Recitation last week had an overview of what it is doing
  - Autograder is posted
- 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.
- Checkin released, due before Wednesday's lecture

Poll Everywhere

pollev.com/tqm

- Any questions?

Lecture Outline

- **Project Overview**
- Refresher
  - stdin, stdout, stderr & File Descriptors
  - Exec
- Pipe
- Unix Shell

Project: Multi-threaded Search Server

- Components:
  - Read files and store them into an index
  - Setup a TCP Server Socket
  - Read & Parse HTTP Requests
  - Handle HTTP Requests & send the appropriate response back

- Demo:
  - Setting up
  - Searching
  - URL & URI

If you normally only look at the slides, you should probably watch this part of the lecture recording.

Lecture Outline

- Project Overview
- **Refresher**
  - **stdin, stdout, stderr & File Descriptors**
  - **Exec**
- Pipe
- Unix Shell

Lecture Outline

- C++ & C Interop
- Processes & Fork
- **stdin, stdout, stderr & File Descriptors**
- Exec
- Pipe

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`

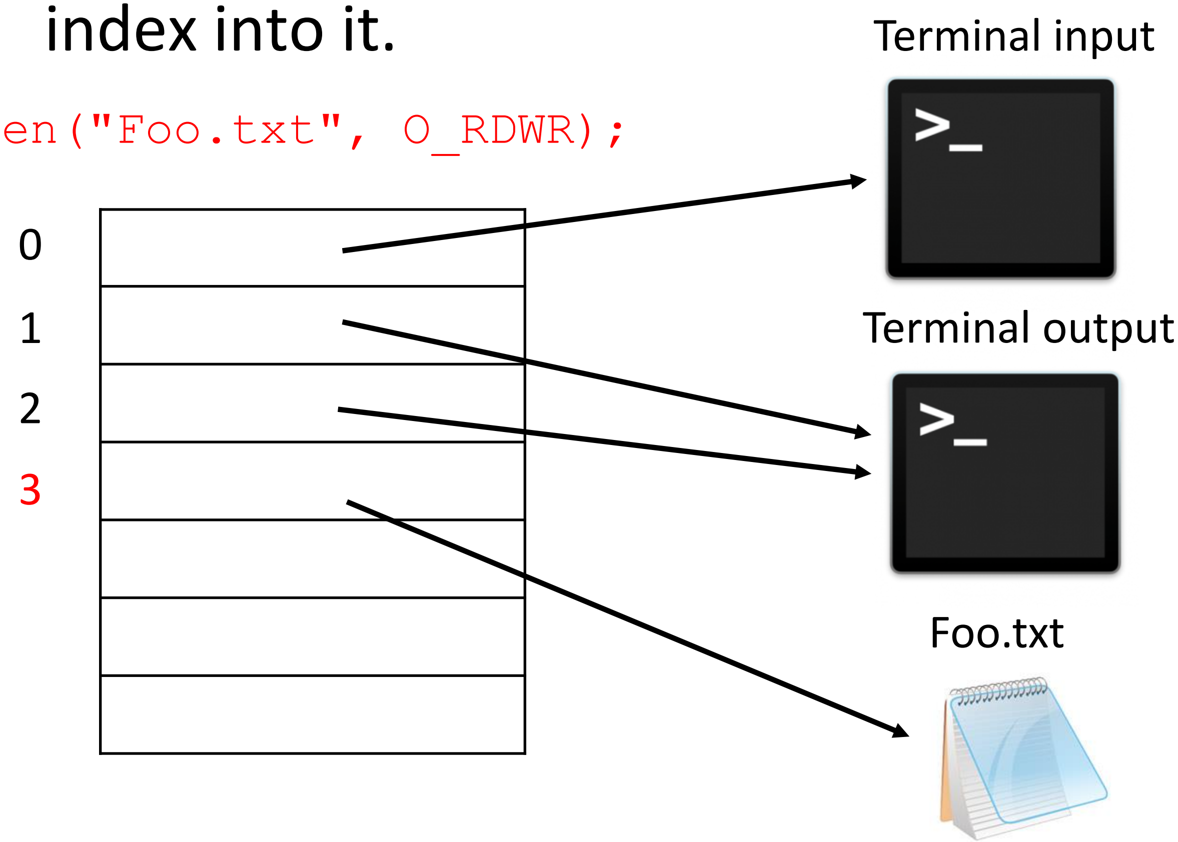
- These will be open on default for a process

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

File Descriptor Table

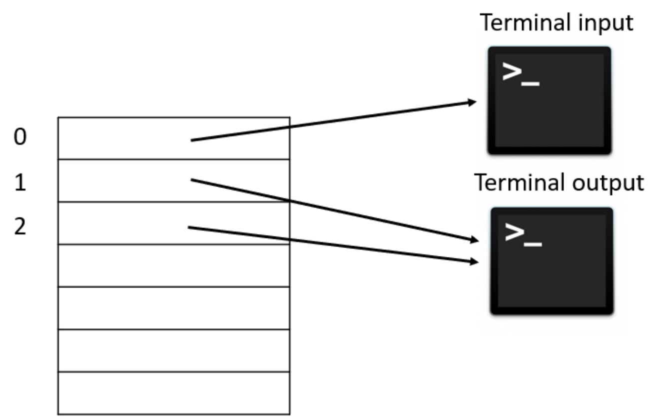
- In addition to an address space, each process will have **its own file descriptor table** managed by the OS
- The table is just an array, and the file descriptor is an index into it.

```
open("Foo.txt", O_RDWR);
```



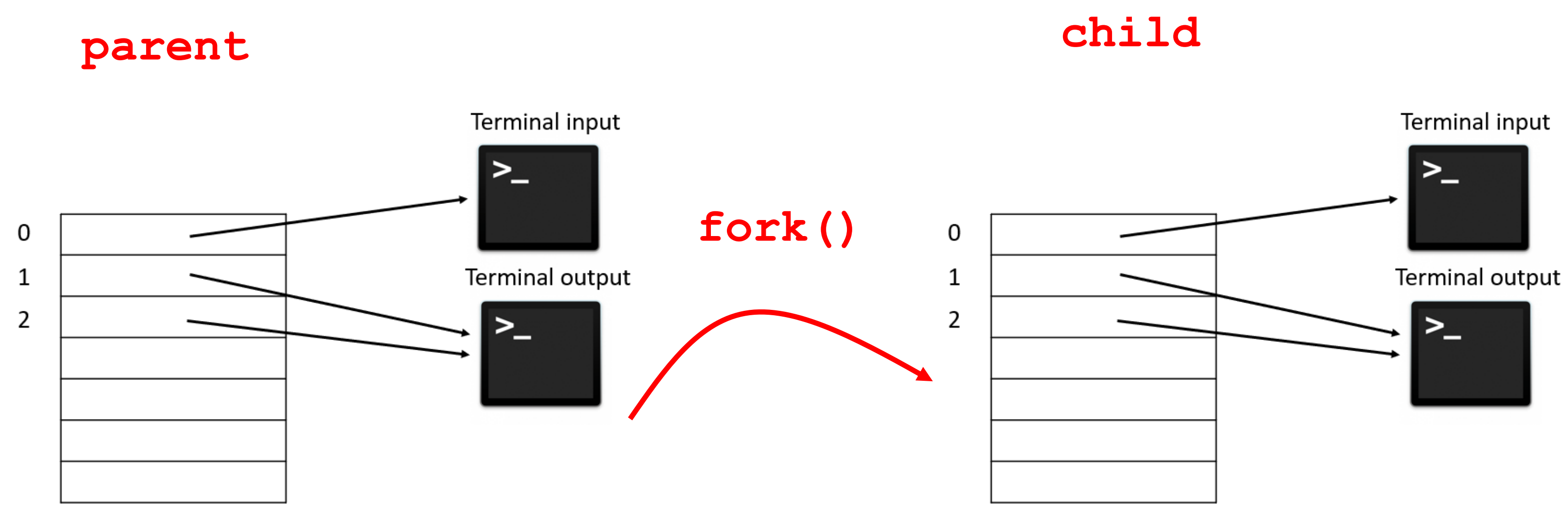
File Descriptor Table: Per Process

- 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



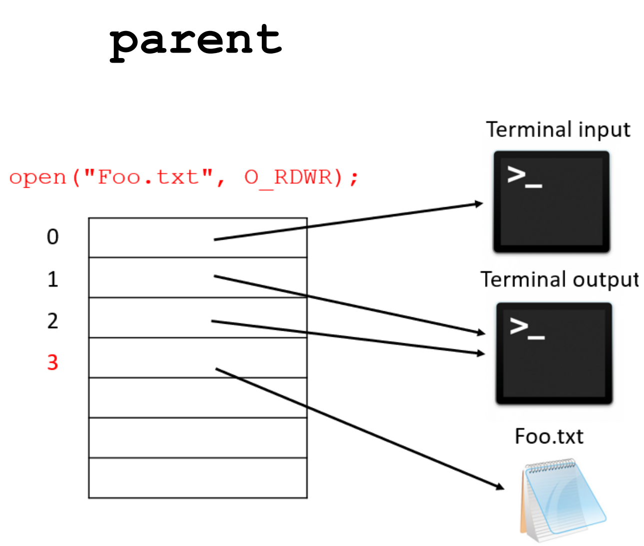
File Descriptor Table: Per Process

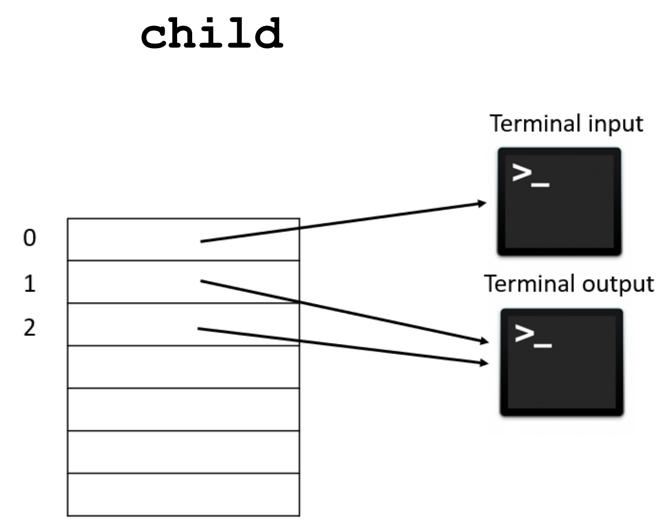
- each process will have **its own file descriptor table** managed by the OS
- Fork will make a copy of the parent's file descriptor table for the child



File Descriptor Table: Per Process

- each process will have **<u>its own file descriptor table</u>** 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!

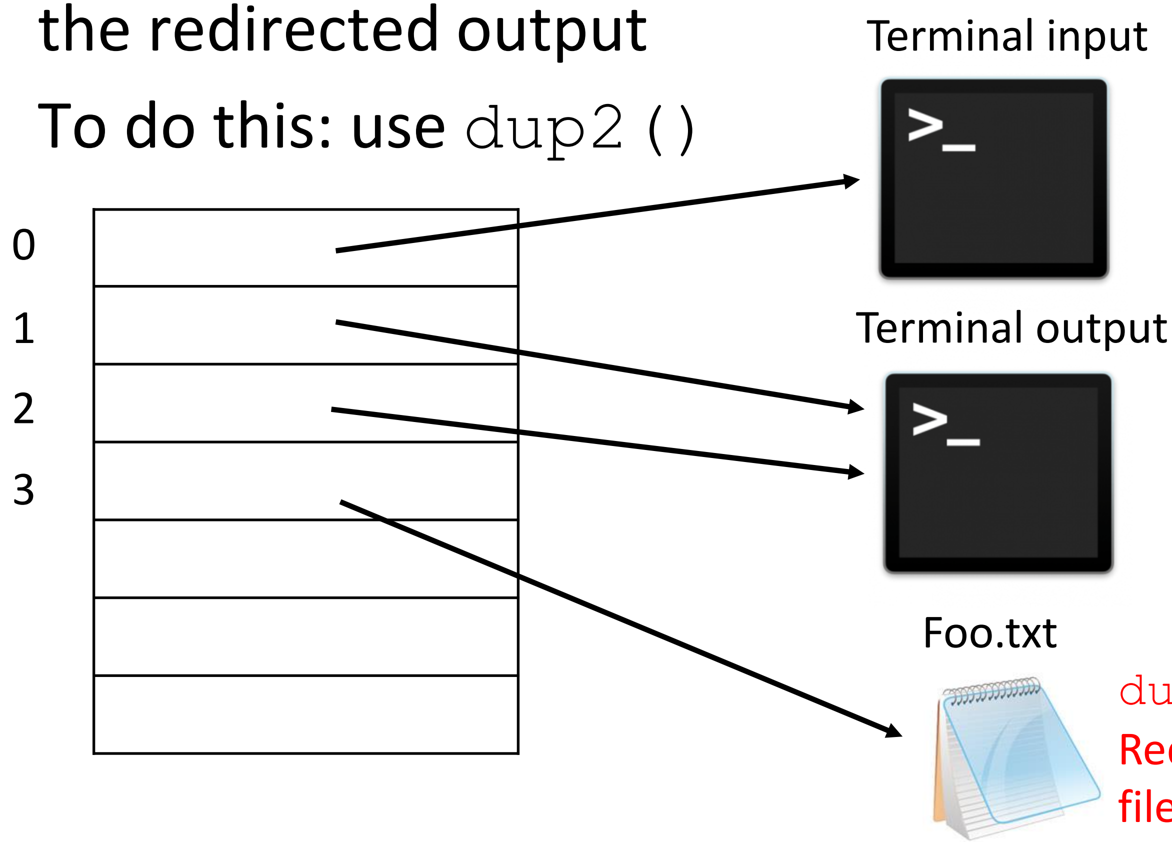
Gap Slide

- 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

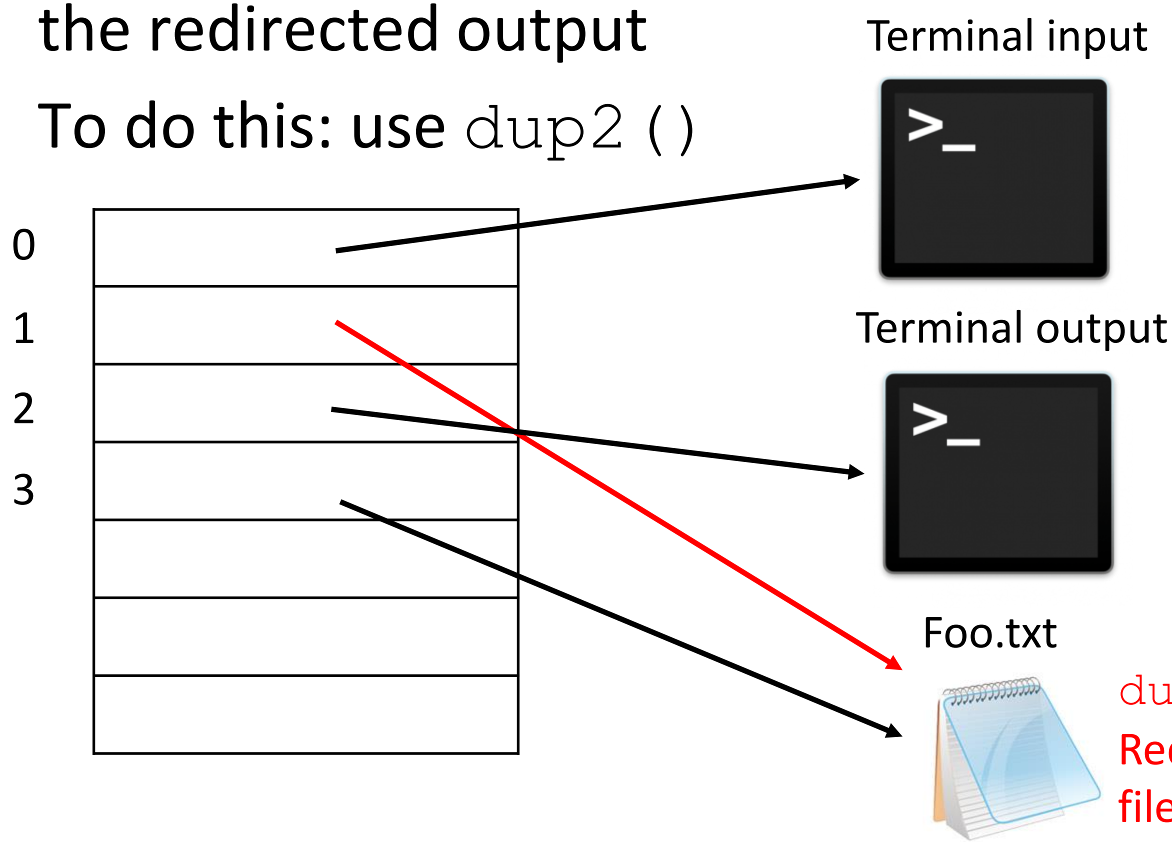
- We can change things so that STDOUT_FILENO is associated with something other than a terminal output.
- Now, any calls to printf, cout, System.out, etc now go to the redirected output
- To do this: use `dup2()`



`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()`

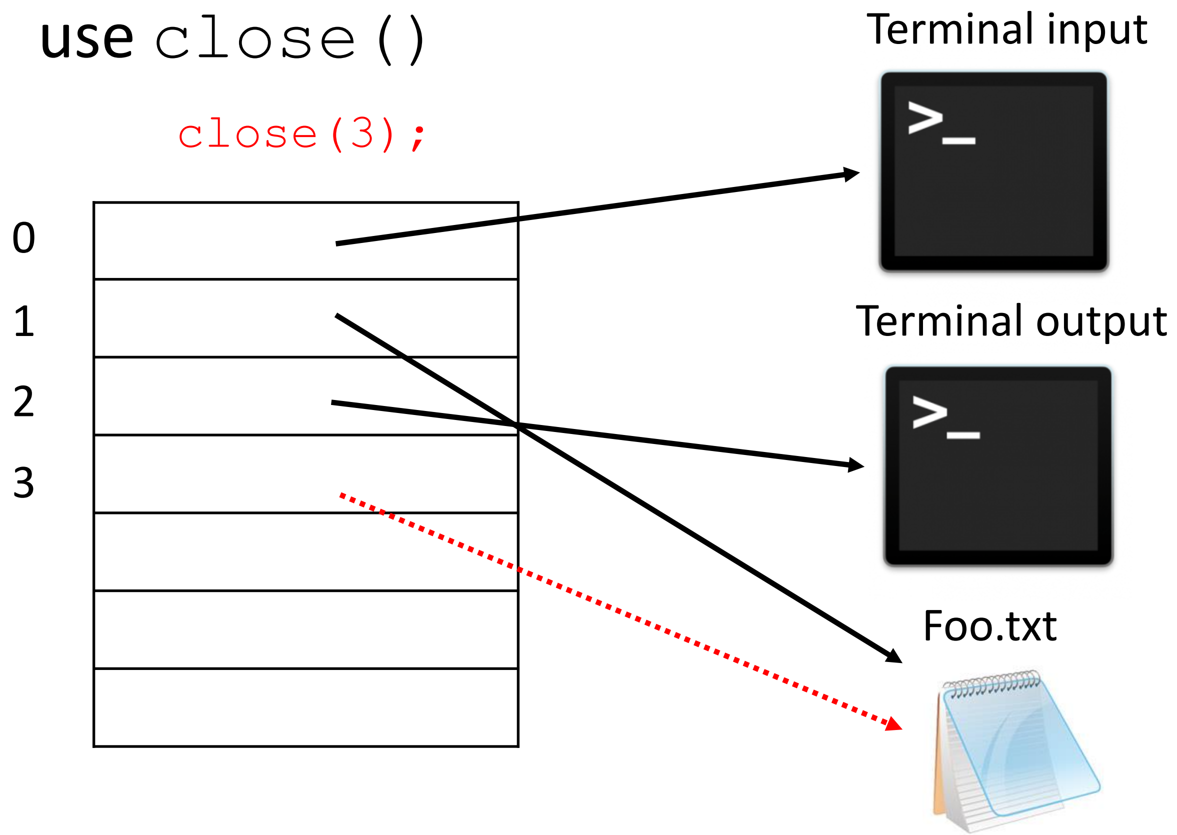


`dup2(3, STDOUT_FILENO);`
Redirects stdout to go to file descriptor 3's destination

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()`

```
close(3);
```



exec*()

- Loads in a new program for execution

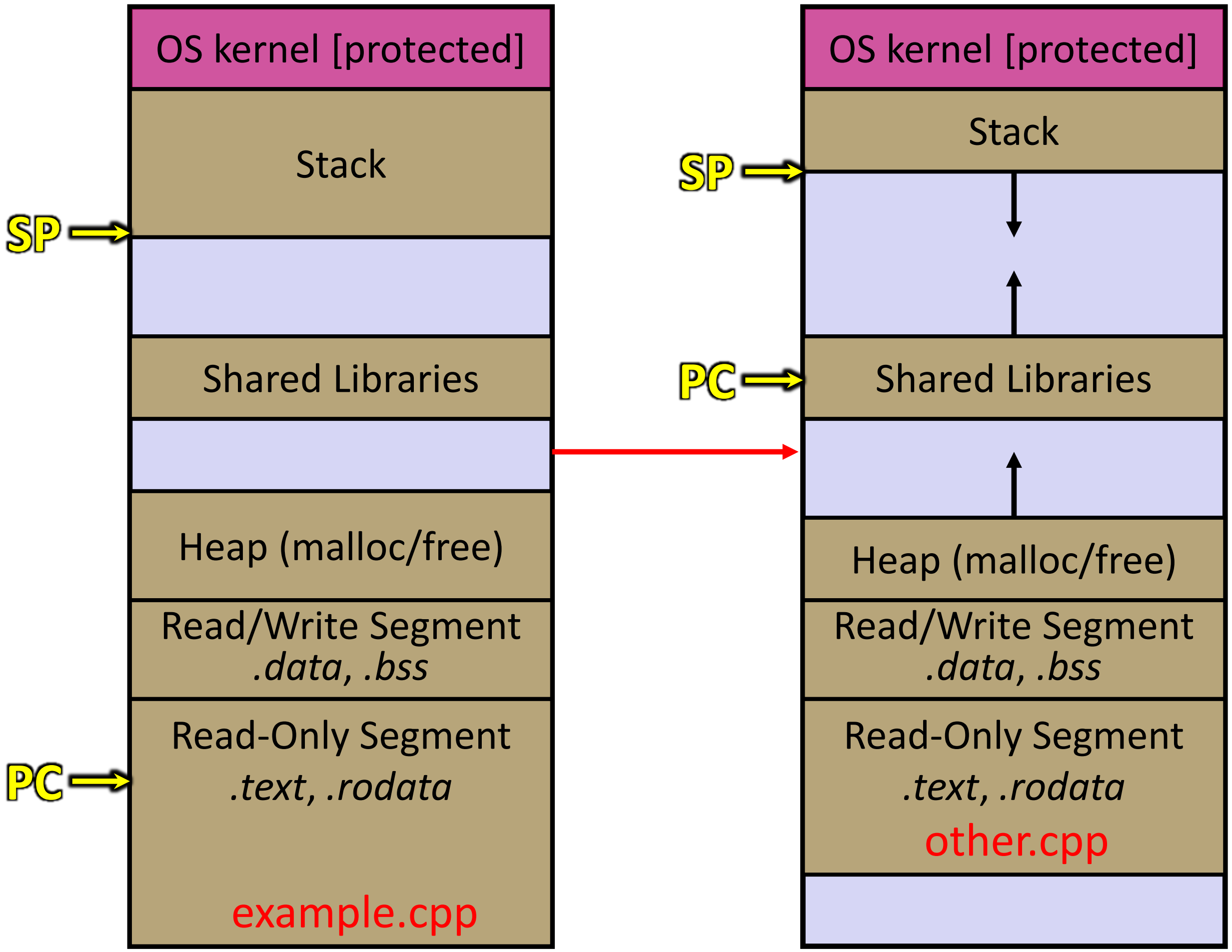
- PC, SP, registers, and memory are all reset so that the specified program can run

execvp()

- ```
  int execvp(const char *file,
                  char* const argv[]);
  ```
- Duplicates the action of the shell (terminal) in terms of finding the command/program to run
- Argv is an array of **`char*`**, the same kind of argv that is passed to `main()` in a C/C++ program
  - **`argv[0]`** MUST have the same contents as the file parameter
  - **`argv`** must have NULL/nullptr as the last entry of the array
- Returns -1 on error. Does NOT return on success
  ```

Exec Visualization

- Exec takes a process and discards or "resets" most of it



NOTE that the following DO change
- The stack
- The heap
- Globals
- Loaded code
- Registers

NOTE that the following do NOT change
- Process ID
- Open files
- The kernel

Exec Demo

- See `exec_example.cpp`
  - Brief code demo to see how exec works
  - What happens when we call exec?
  - What happens if we open some files before exec?
  - What happens if we replace stdout with a file?

- NOTE: When a process exits, then it will close all of its open files by default

Poll Everywhere

pollev.com/tqm

```
int main(int argc, char* argv[]) {
  // fork a process to exec clang
  pid_t clang_pid = fork();
  if (clang_pid == 0) {
    // we are the child
    char* clang_argv[] = {"g++-12", "-o",
                 "hello","hello.cpp", NULL};
    execvp(clang_argv[0], clang_argv);
    exit(EXIT_FAILURE);
  }

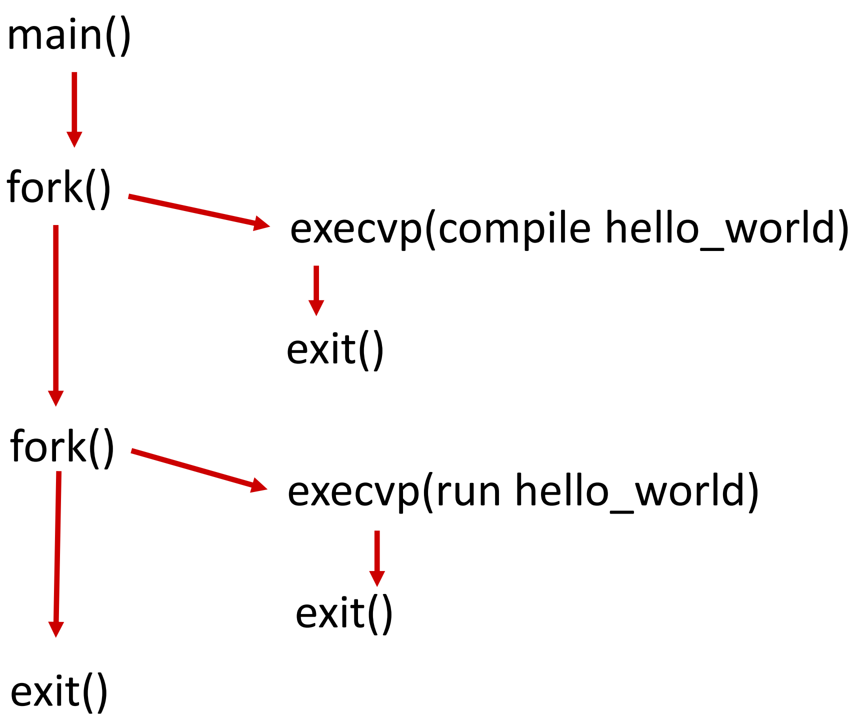
  // fork to run the compiled program
  pid_t hello_pid = fork();
  if (hello_pid == 0) {
    // the process created by fork
    char* hello_argv[] = {"./hello", NULL};
    execvp(hello_argv[0], hello_argv);
    exit(EXIT_FAILURE);
  }
  return EXIT_SUCCESS;
}
```

This code is broken. It compiles, but it doesn't do what we want. Why?

- g++-12 is a C++ compiler
- I want to compile and run hello.cpp
- Assume it compiles
- Assume I gave the correct args to exec

Poll Everywhere

pollev.com/tqm



This code is broken. It compiles, but it doesn't do what we want. Why?

- g++-12 is a C++ compiler
- I want to compile and run hello.cpp
- Assume it compiles
- Assume I gave the correct args to exec

Lecture Outline

- Project Overview
- Refresher
  - stdin, stdout, stderr & File Descriptors
  - Exec
- **Pipe**
- Unix Shell

Pipes

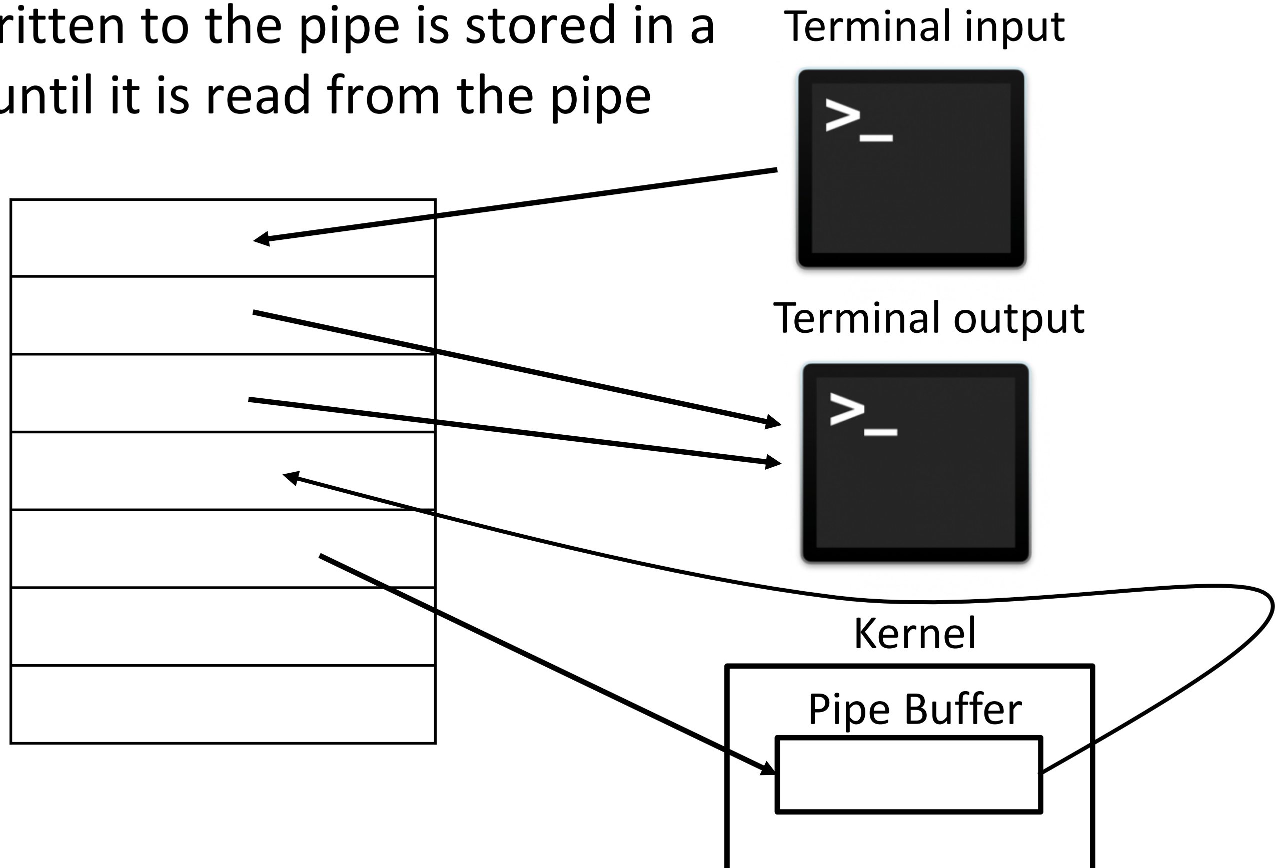
```
int pipe(int pipefd[2]);
```

- 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**

Poll Everywhere

pollev.com/tqm

- What does the parent print? What does the child print? why? (assume pipe, close and fork succeed)

```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);

int main() {
  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[0]);

    string greeting {"Hello!"};
    wrapped_write(pipe_fds[1], greeting);

    optional<string> response = wrapped_read(pipe_fds[1]);

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

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

pipe_unidirect.cpp
on course website

```cpp
  // parent

  /// close the end of the pipe I won't use
  close(pipe_fds[1]);

  optional<string> message = wrapped_read(pipe_fds[0]);

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

  string greeting{"Howdy!"};
  wrapped_write(pipe_fds[0], greeting);

  int wstatus;
  waitpid(pid, &wstatus, 0);

  return EXIT_SUCCESS;
}
```

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

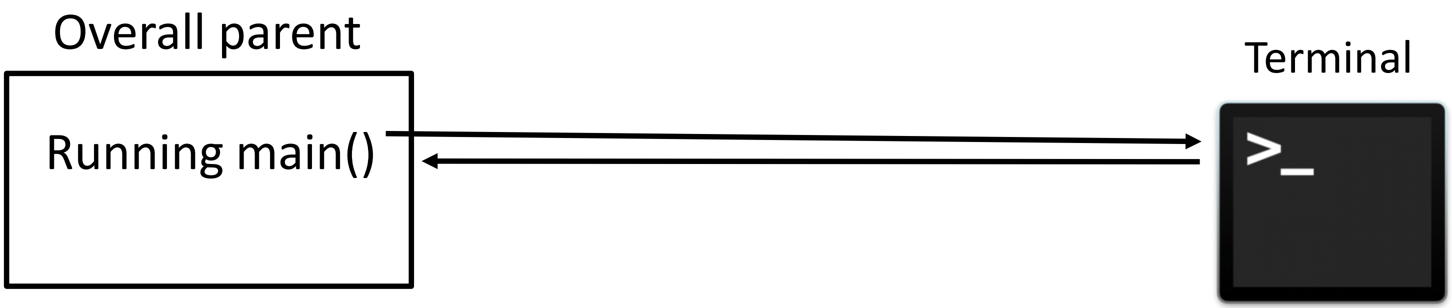
- **Good practice: CLOSE ALL PIPE FDS YOU ARE DONE WITH**

Exec & Pipe Demo

- See `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

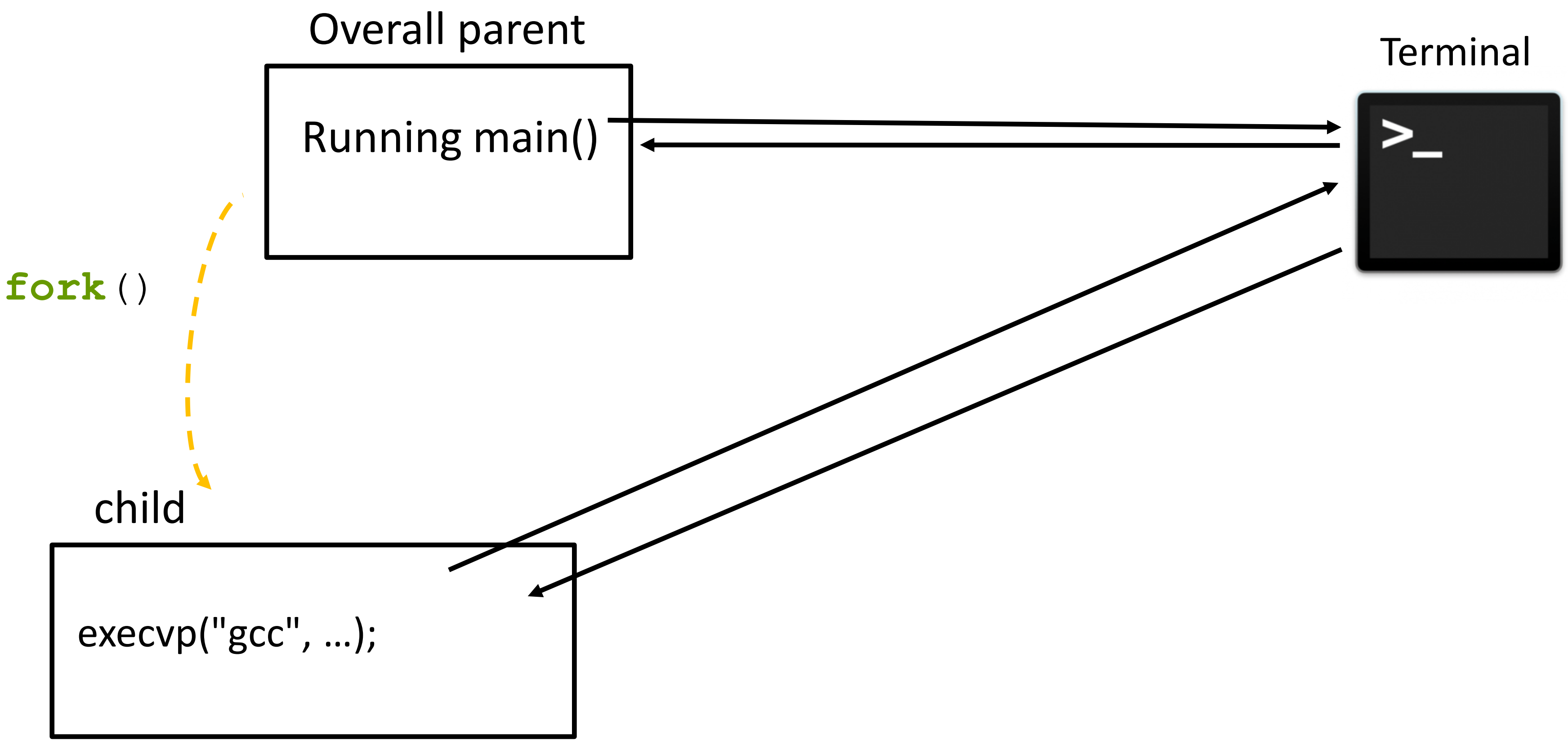
`io_autograder.cpp` Trace

- First:
  we compile the program with the gcc command



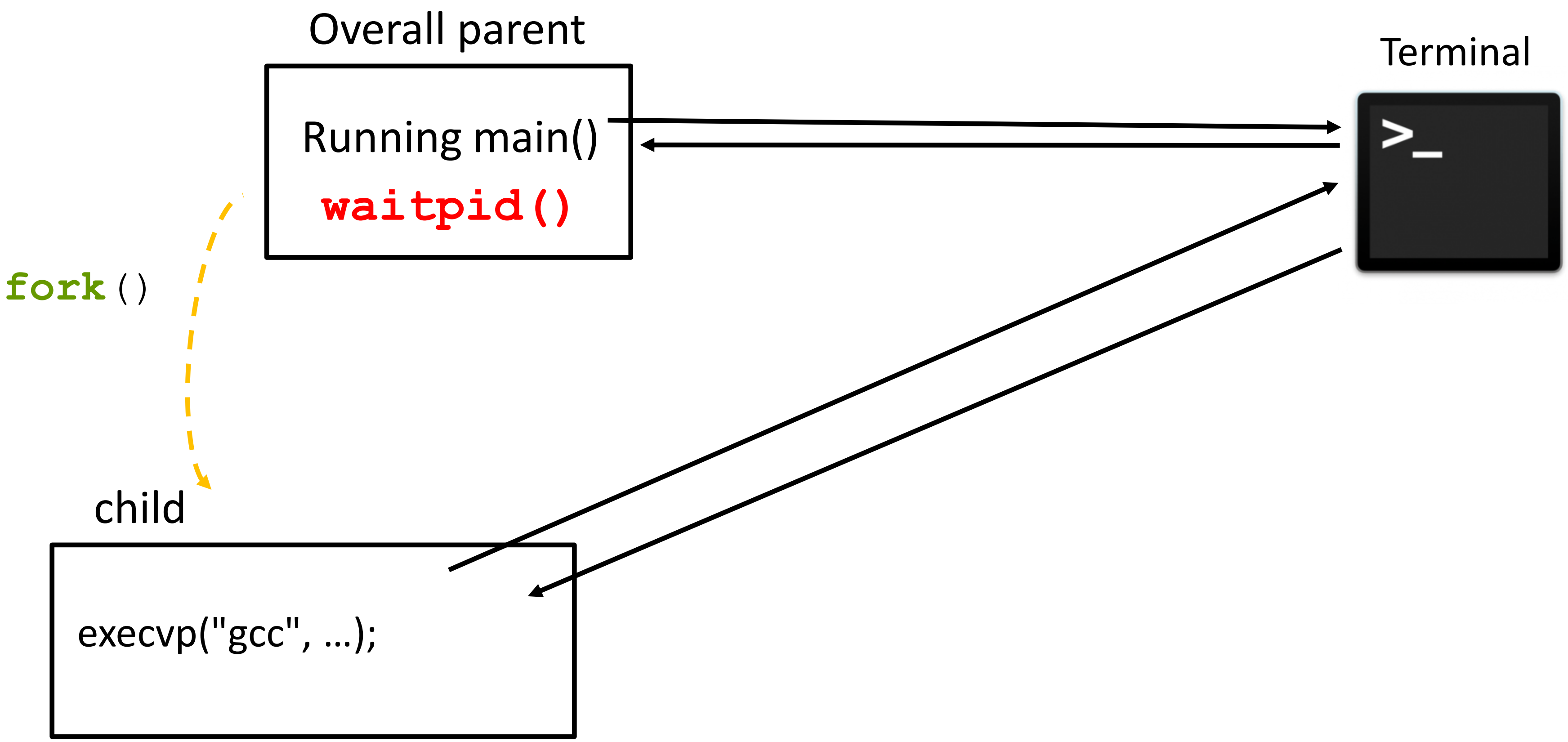
`io_autograder.cpp` Trace

- First:
  we compile the program with the gcc command



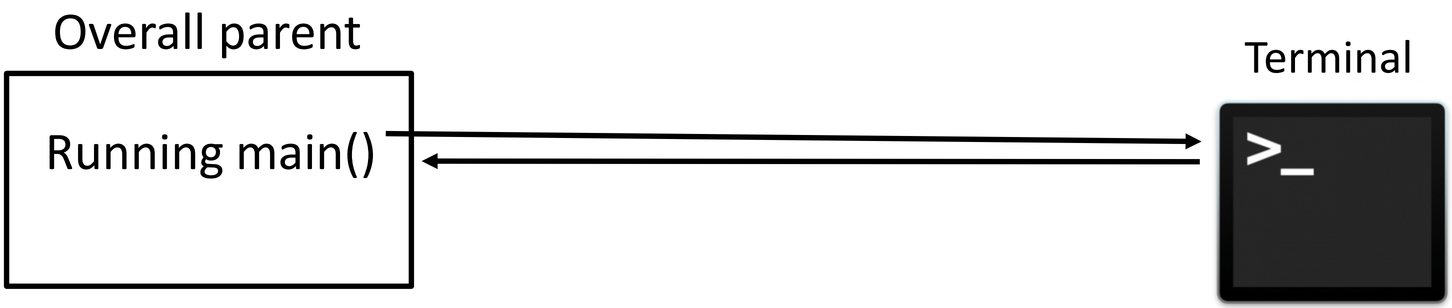
io_autograder.cpp Trace

- First:
  we compile the program with the gcc command



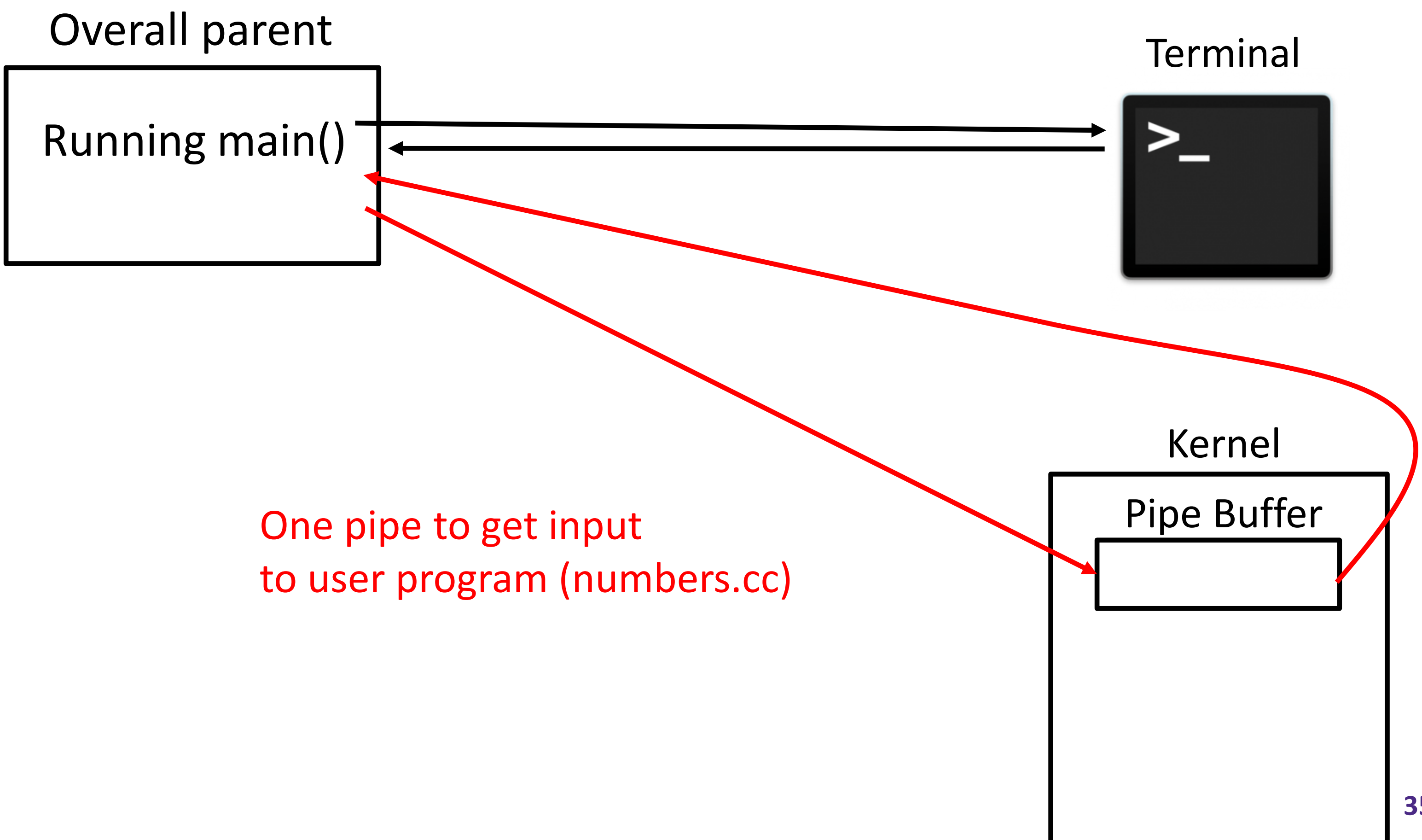
`io_autograder.cpp` Trace

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



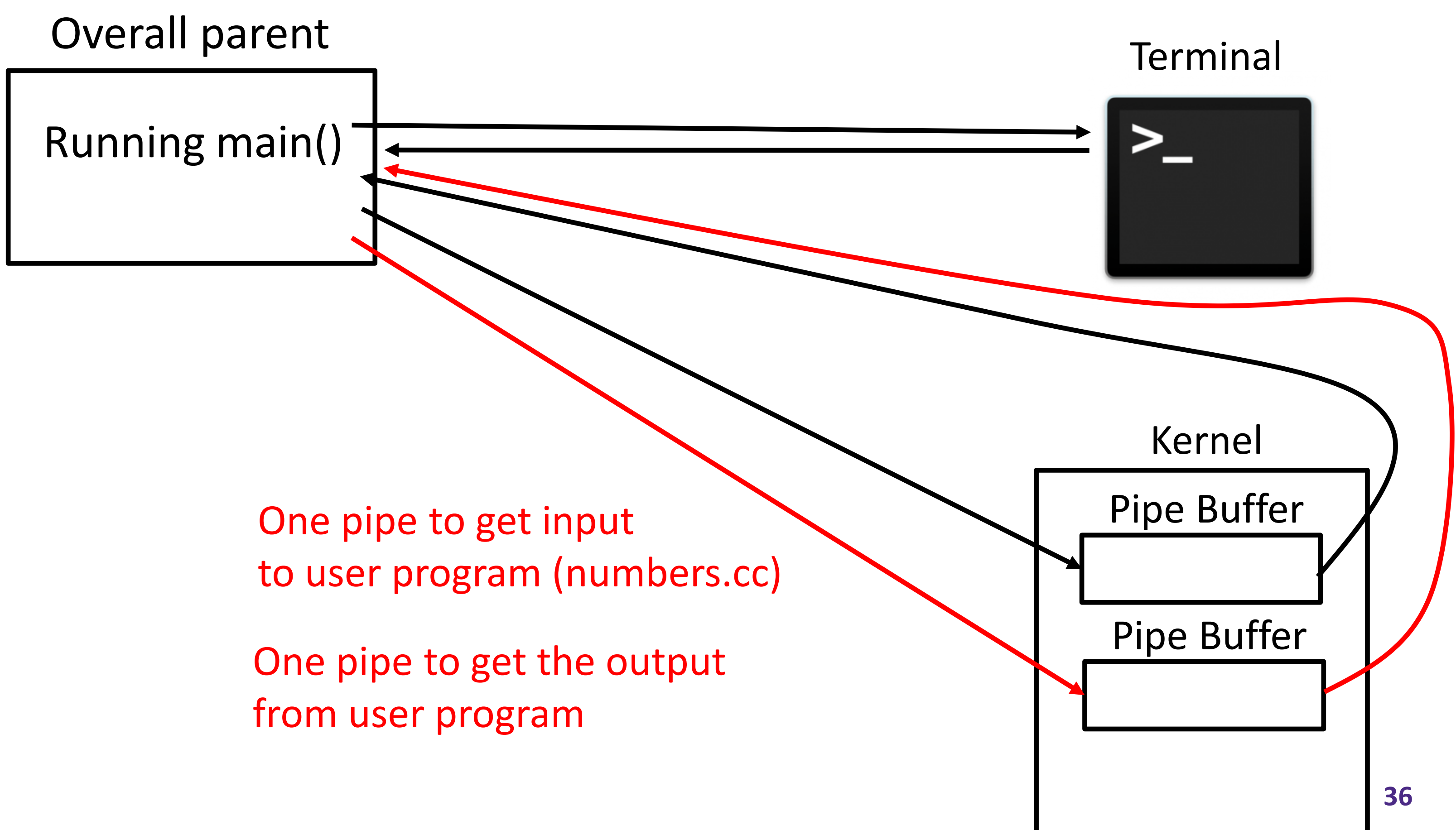
`io_autograder.cpp` Trace

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



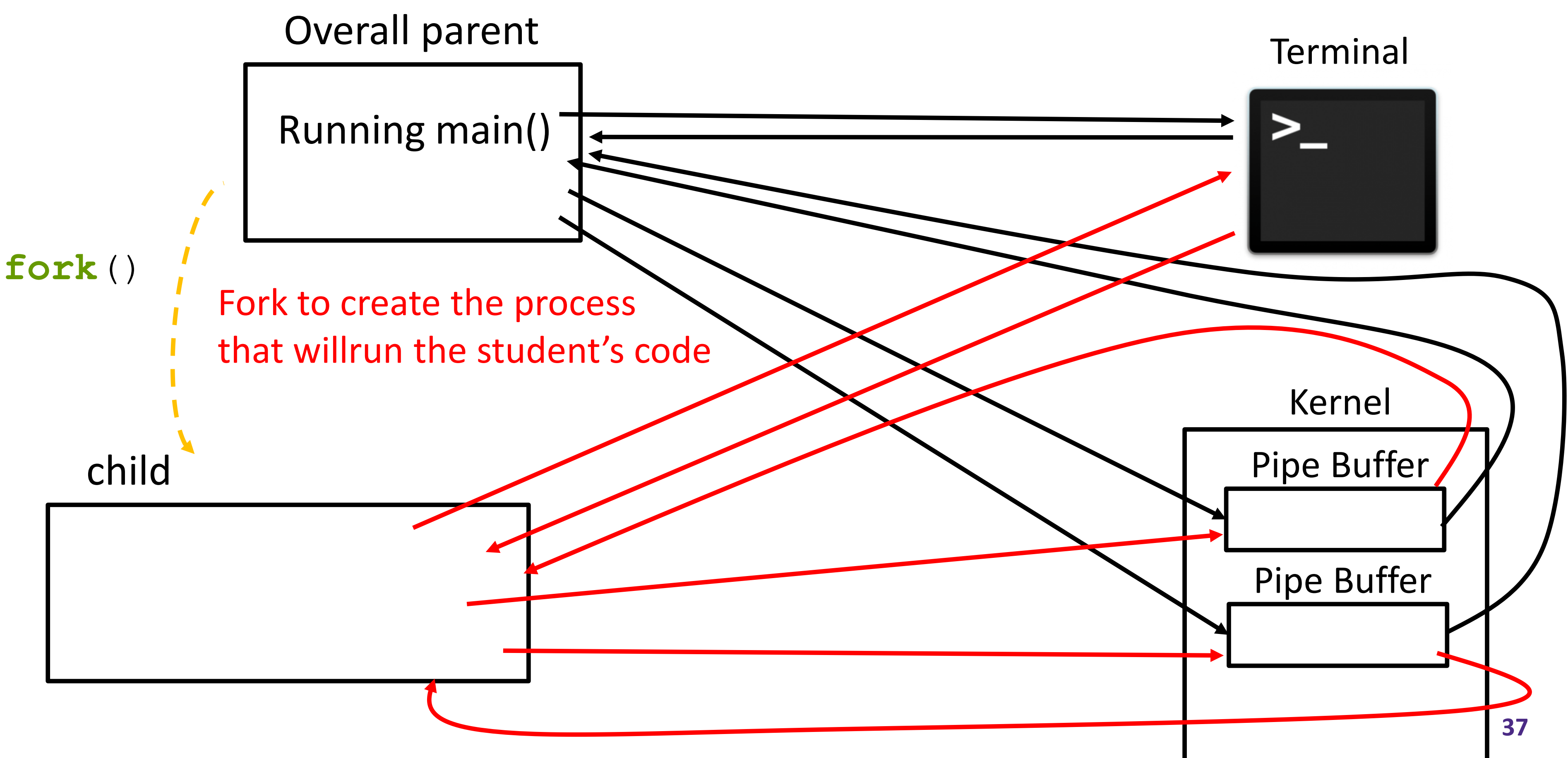
io_autograder.cpp Trace

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



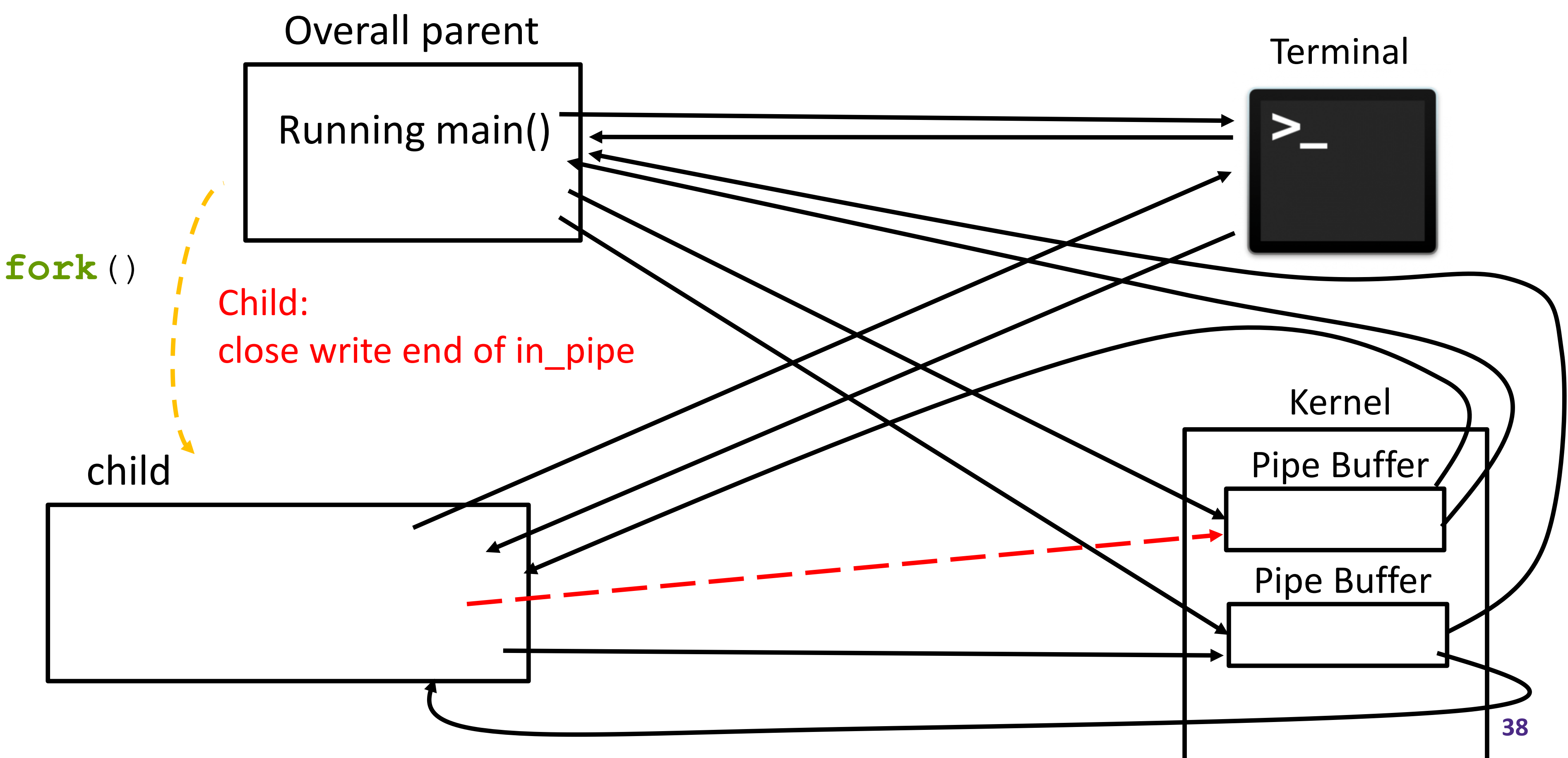
io_autograder.cpp Trace

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



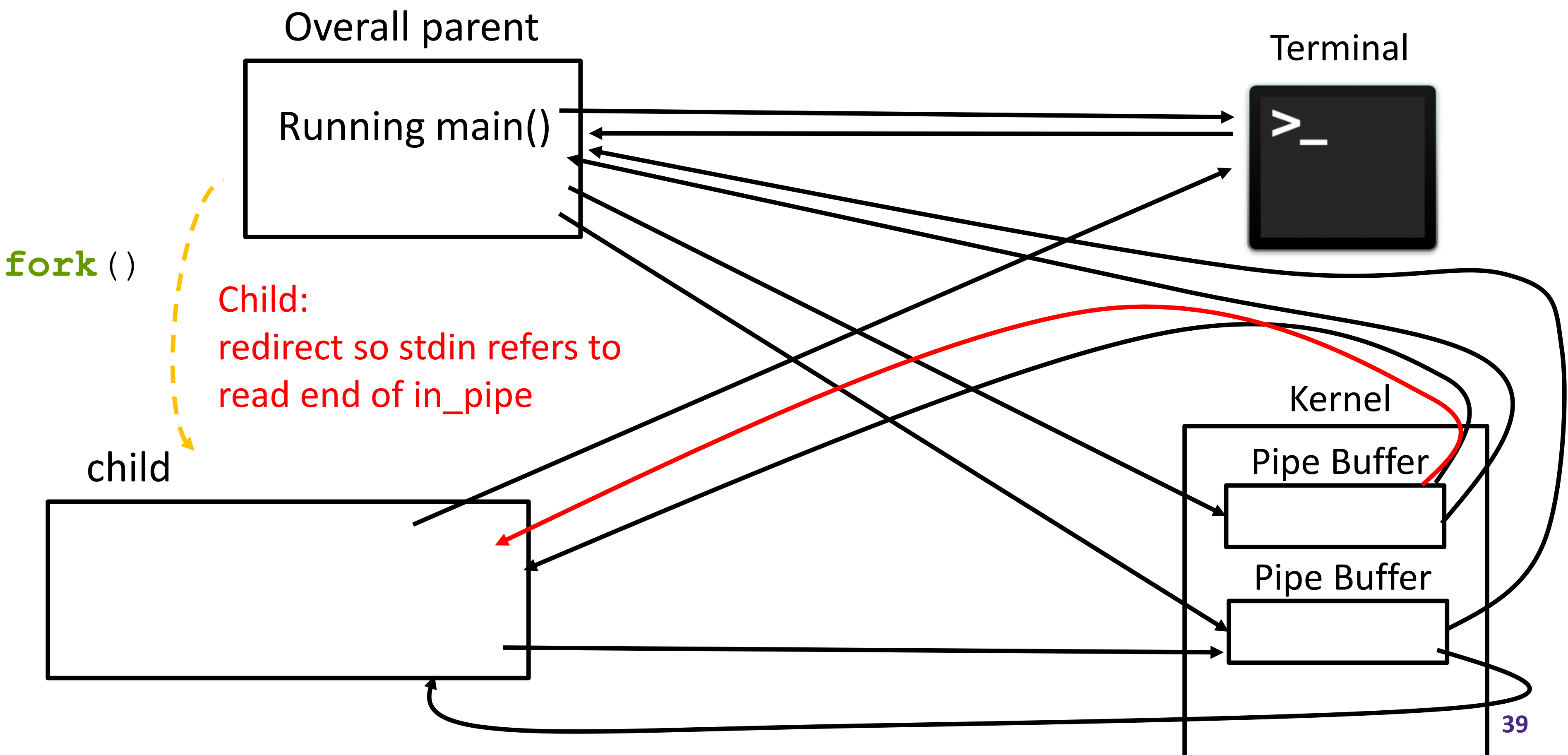
`io_autograder.cpp` Trace

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



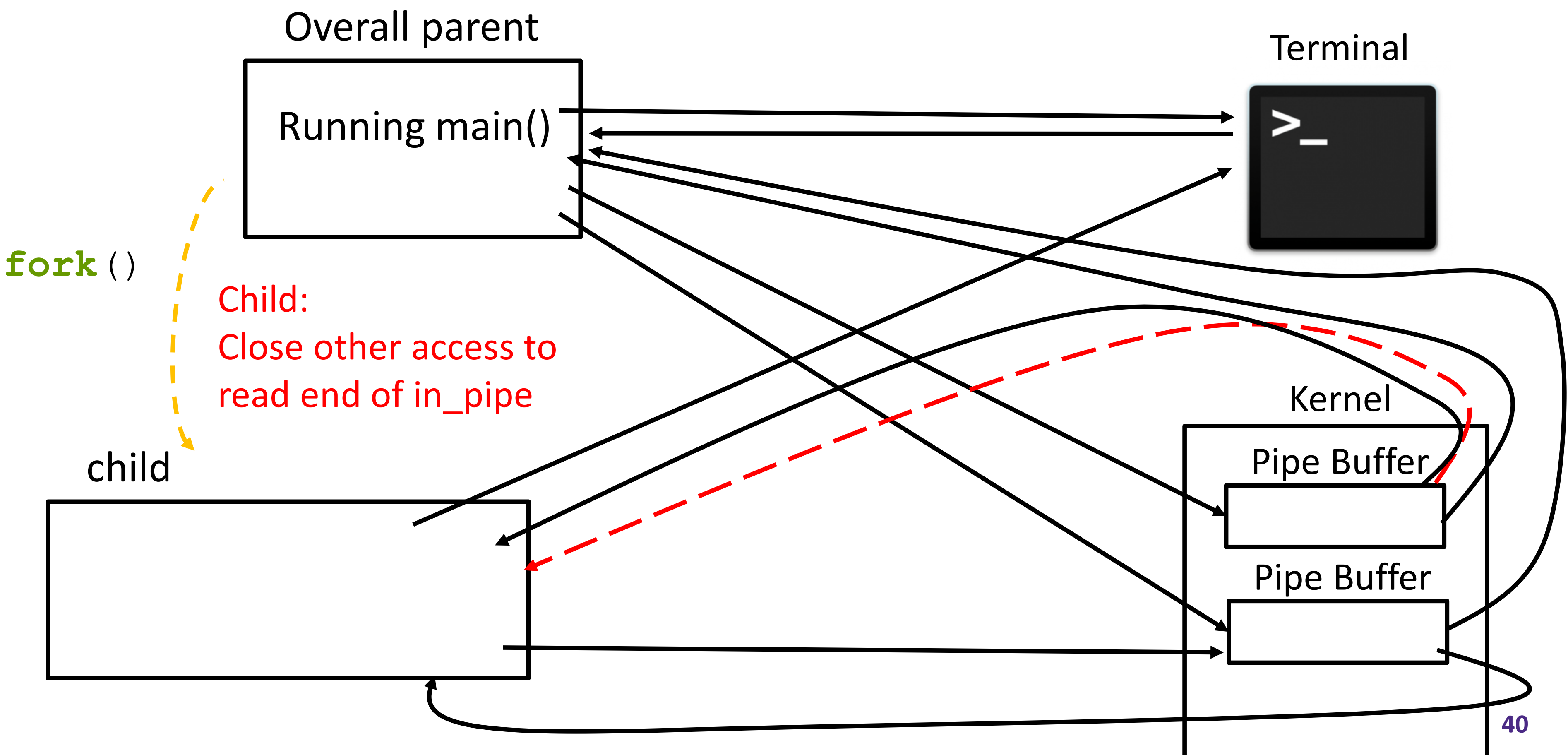
io_autograder.cpp Trace

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



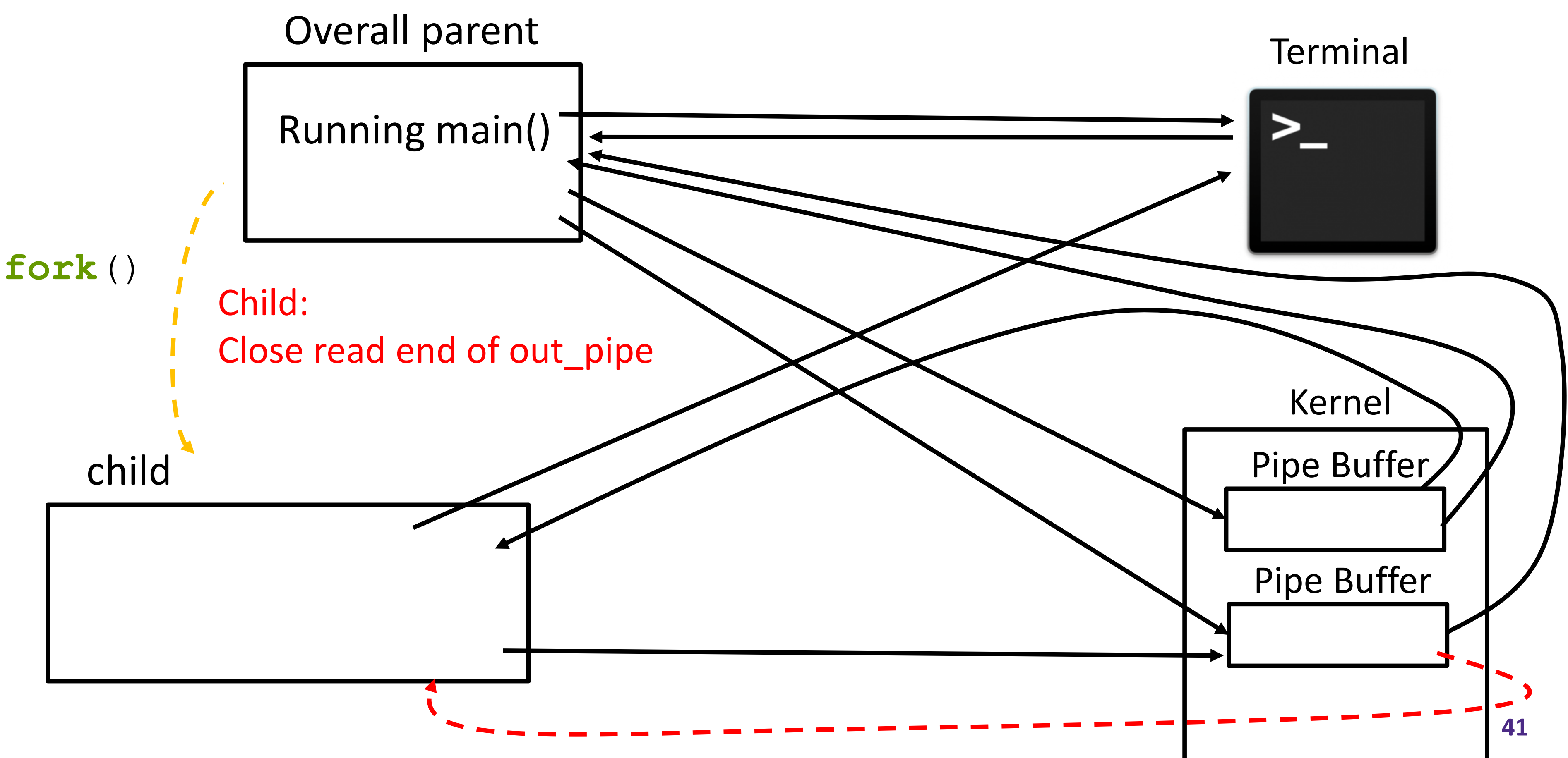
`io_autograder.cpp` Trace

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



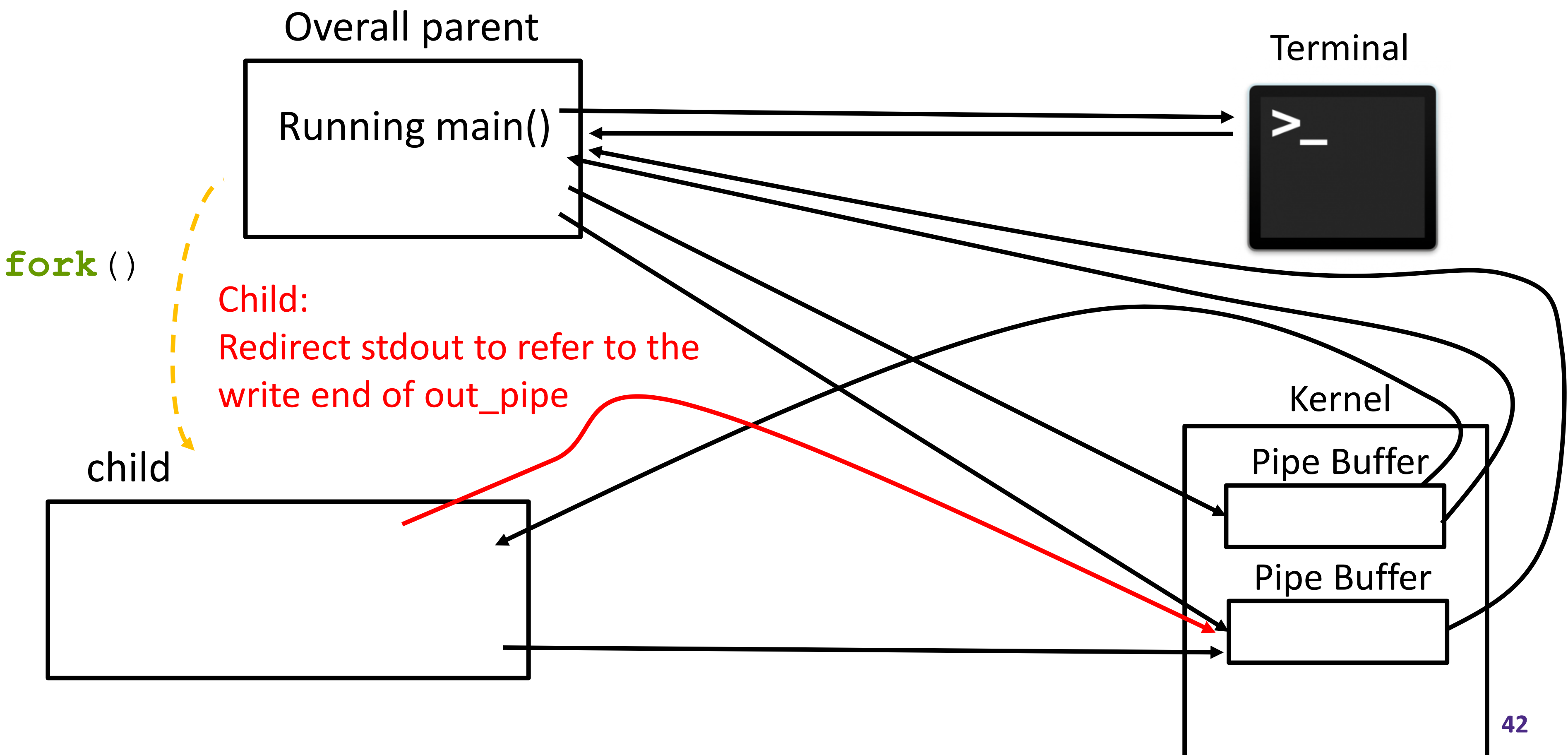
io_autograder.cpp Trace

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



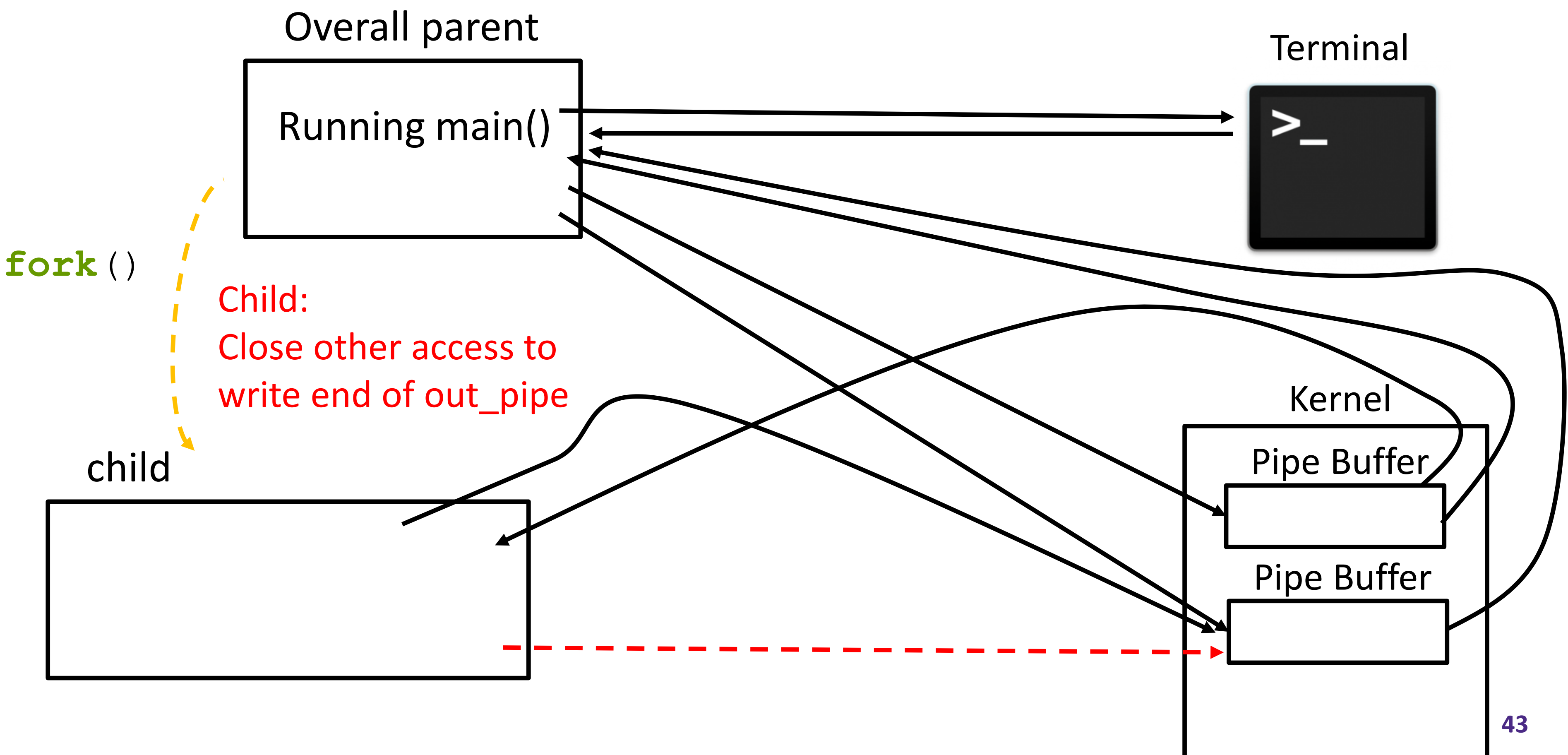
`io_autograder.cpp` Trace

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



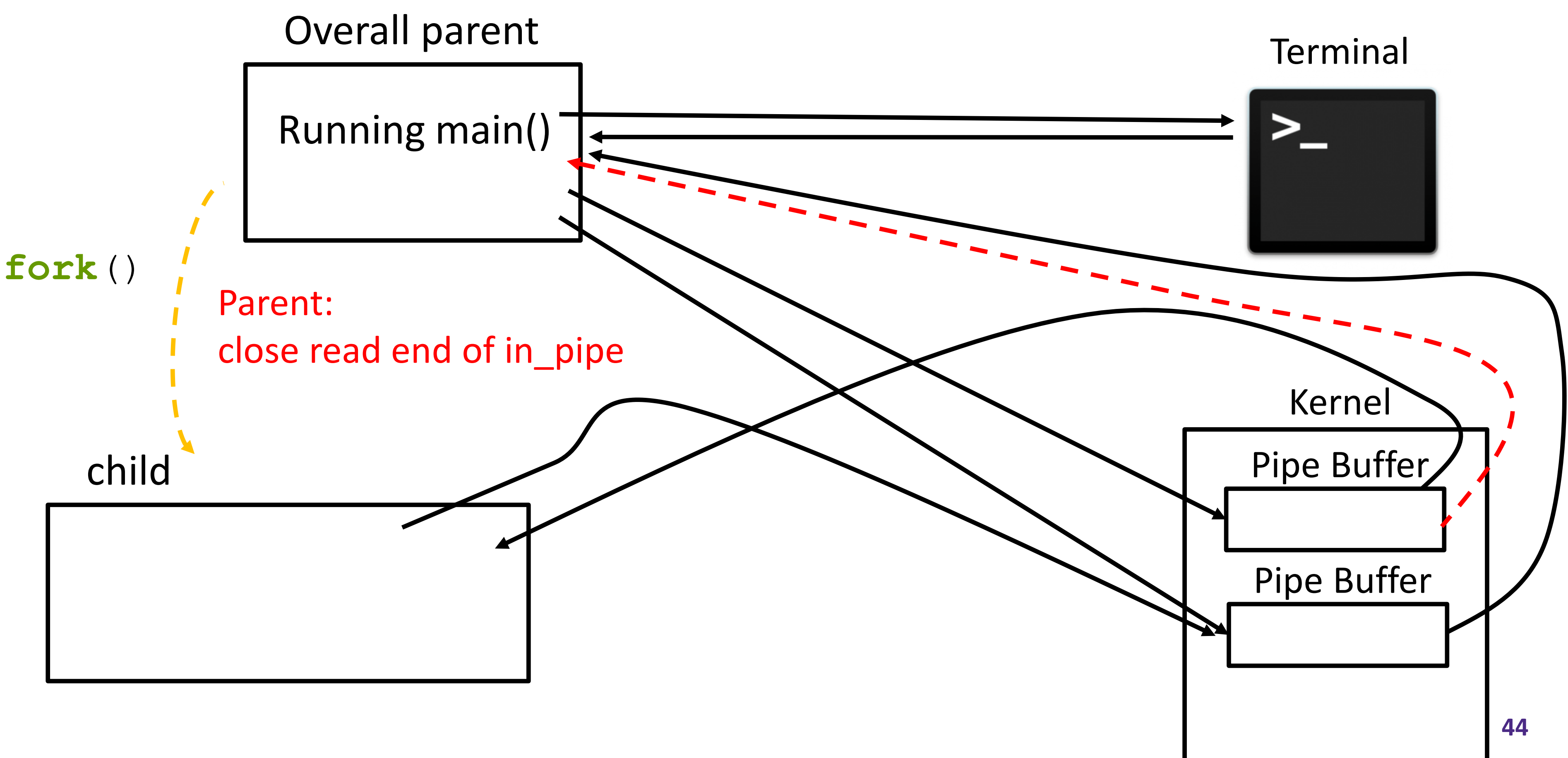
`io_autograder.cpp` Trace

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



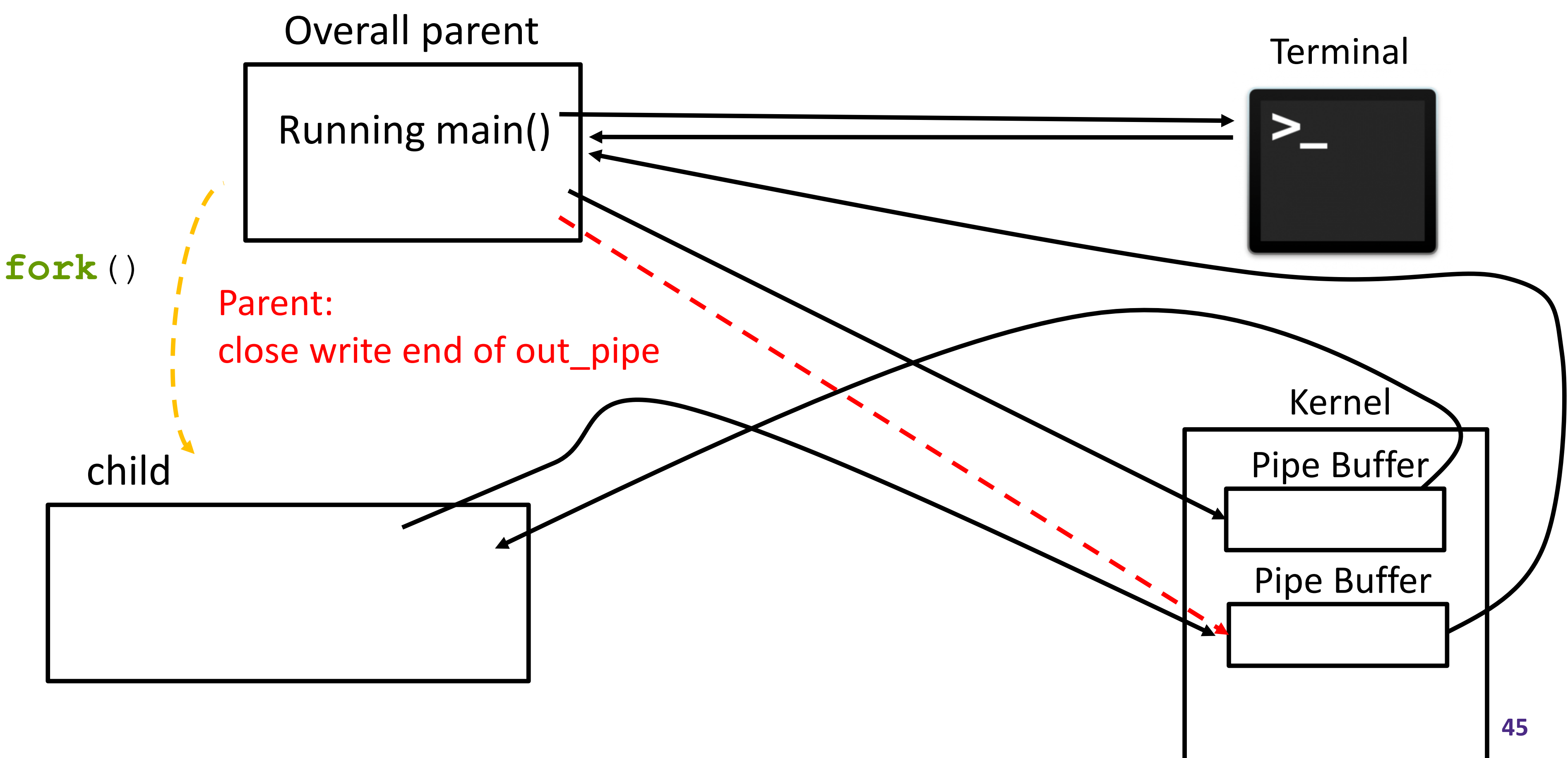
io_autograder.cpp Trace

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



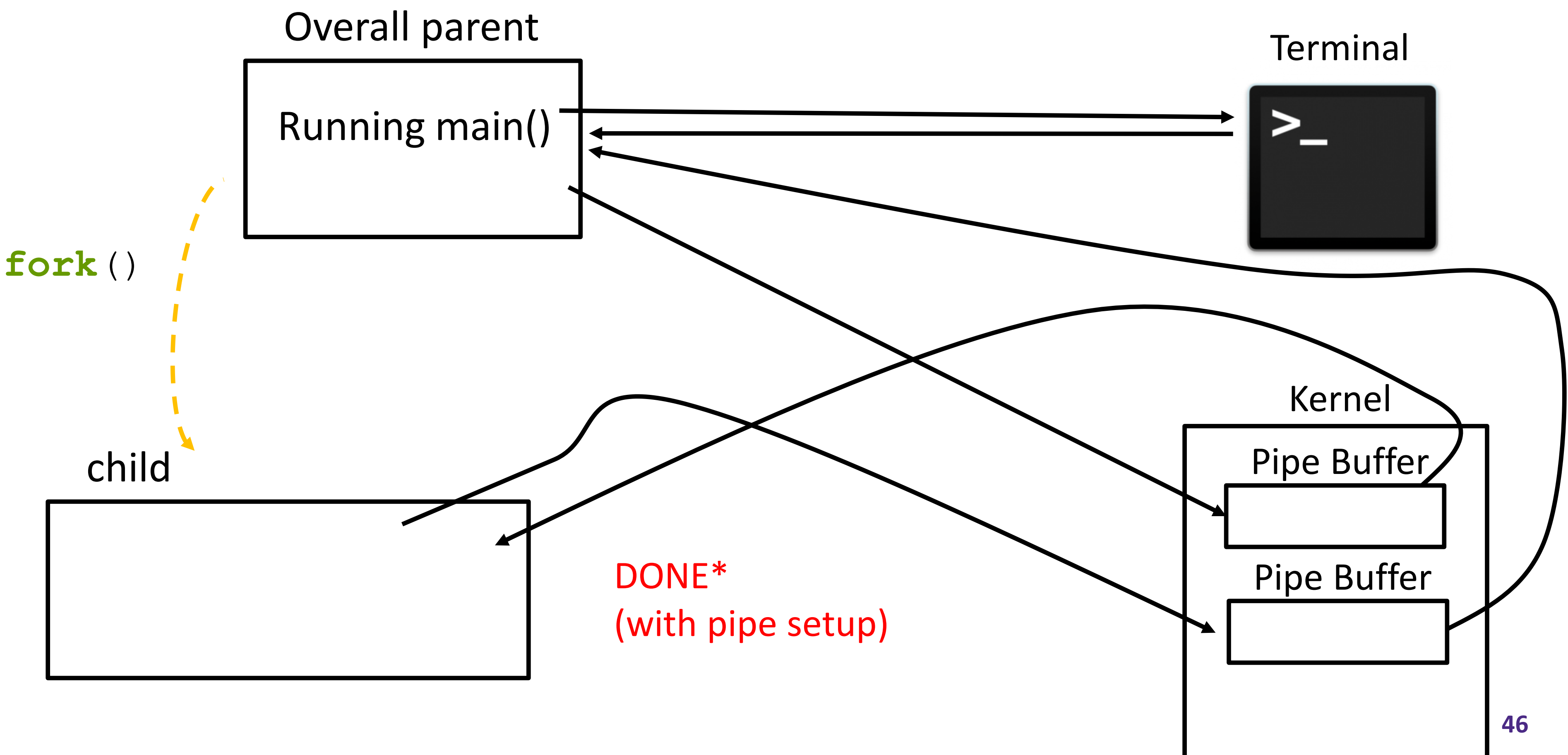
`io_autograder.cpp` Trace

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



io_autograder.cpp Trace

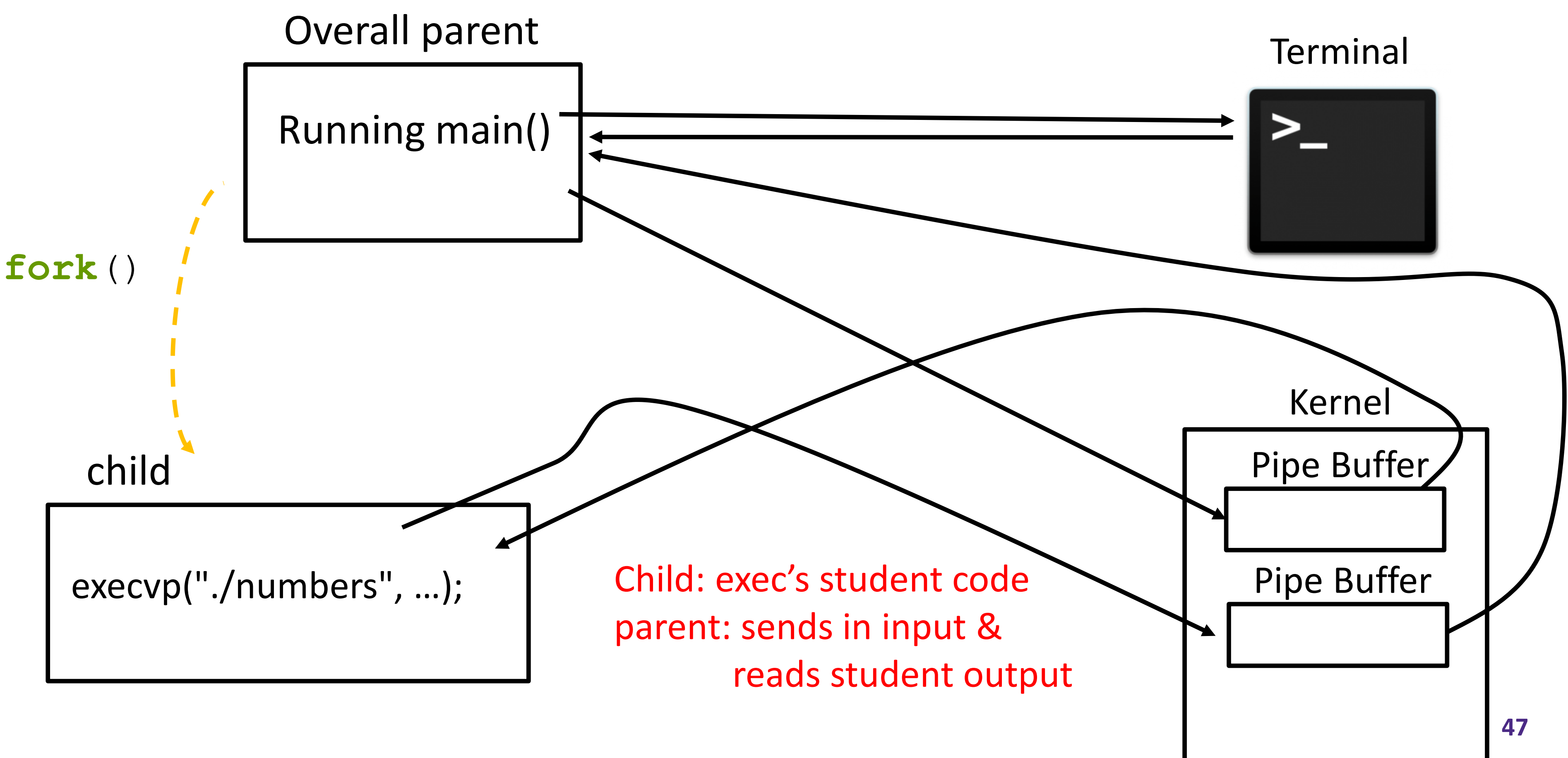
- Compilation done! Run the compiled program… BUT send autograder input and capture output



DONE*
(with pipe setup)

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

- Project Overview
- Refresher
  - stdin, stdout, stderr & File Descriptors
  - Exec
- Pipe
- **Unix Shell**

Unix Shell

- A **<u>user level</u>** 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

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.

That's all!

- More on pipe in next lecture!

- Any questions?