Pipes & File Descriptors

Computer Operating Systems, Spring 2024

Instructor: Travis McGaha

Head TAs: Nate Hoaglund & Seungmin Han

TAs:

Adam Gorka	Haoyun Qin	Kyrie Dowling	Ryoma Harris
Andy Jiang	Jeff Yang	Oliver Hendrych	Shyam Mehta
Charis Gao	Jerry Wang	Maxi Liu	Tom Holland
Daniel Da	Jinghao Zhang	Rohan Verma	Tina Kokoshvili
Emily Shen	Julius Snipes	Ryan Boyle	Zhiyan Lu

pollev.com/tqm

- How confident do you feel in your penn-parser? Would you feel comfortable expanding on it slightly and/or using it in penn-shell (the next homework to release)?

Administrivia

- Project 0 penn-parser:
  - Autograder is up,
- Project 1 penn-shredder:
  - You need penn-parser to complete it
  - Is not much more once you have implemented penn-parser
  - Should have everything you need
- BOTH were extended:
  - **Extended to Monday 2/5 @11:59 pm**

Administrivia

- There will be a check-in due next week
  - Due before Tues @ 5pm
- First “recitation” was yesterday, there will be another again next week
  - Monday @ 3:30 in Moore 100B
  - GDB, Valgrind, Penn-shell tips
- Pre-semester survey:
  - Still open till Friday @mindnight
  - Just a short survey
  - Please do it

Penn-Shredder Compatibility

- From the signal(2) man page

```
Portability
    The only portable use of signal() is to set a signal's disposition to SIG_DFL or SIG_IGN.  The semantics when using signal()
    to establish a signal handler vary across systems (and POSIX.1 explicitly permits this variation); do not use  it  for  this
    purpose.
```

- If you want to have better help from TA's put this at the top of your file before you #include anything
  - This *should* get signals to behave as we expect, so TAs can better help
  - If you got it working another way, that is OK. Auto-grader *should* still accept it

```
#ifndef _POSIX_C_SOURCE
#define _POSIX_C_SOURCE 200809L
#endif

#ifndef _DEFAULT_SOURCE
#define _DEFAULT_SOURCE 1
#endif
```

pollev.com/tqm

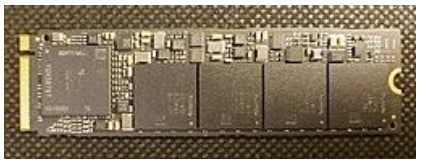
- Any questions regarding penn-parser or penn-shredder?

Lecture Outline

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

What is a File?

- Files are "non-volatile storage" that are external to a process:
  - changes to a file persist beyond the lifetime of a process
  - The same file can be access by multiple processes
  - Stored on completely different hardware than normal process memory



- More details on Files in the second half of the semester

What is a file descriptor?

- A file descriptor is of type `int`
  - A unique id a process can use to refer to a file when invoking system calls
- A file descriptor may not refer to a file, but instead refer to something that is "like a file"
  - Terminal input/output
  - Network connections
  - Pipes (more later this lecture)
  - Special devices
- These can all be used for **`read()`** and **`write()`**

stdout, stdin, stderr

- By default, there are three "files" open when a program starts
  - stdin: for reading terminal input typed by a user
    - `stdin` in C `stdio.h`
    - `System.in` in Java
  - stdout: the normal terminal output. (buffered)
    - `stdout` in C `stdio.h`
    - `System.out` in Java
  - stderr: the terminal output for printing errors (unbuffered)
    - `stderr` in C `stdio.h`
    - `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 `printf` will use `write(STDOUT_FILENO, …)`

open()/close()

- `int open(const char* pathname, int flags);`
  - Pass in the filename and access mode
  - Returns a file descriptor or -1 on error

- `int close(int fd);`
  - Closes specified fd, not the specified file

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

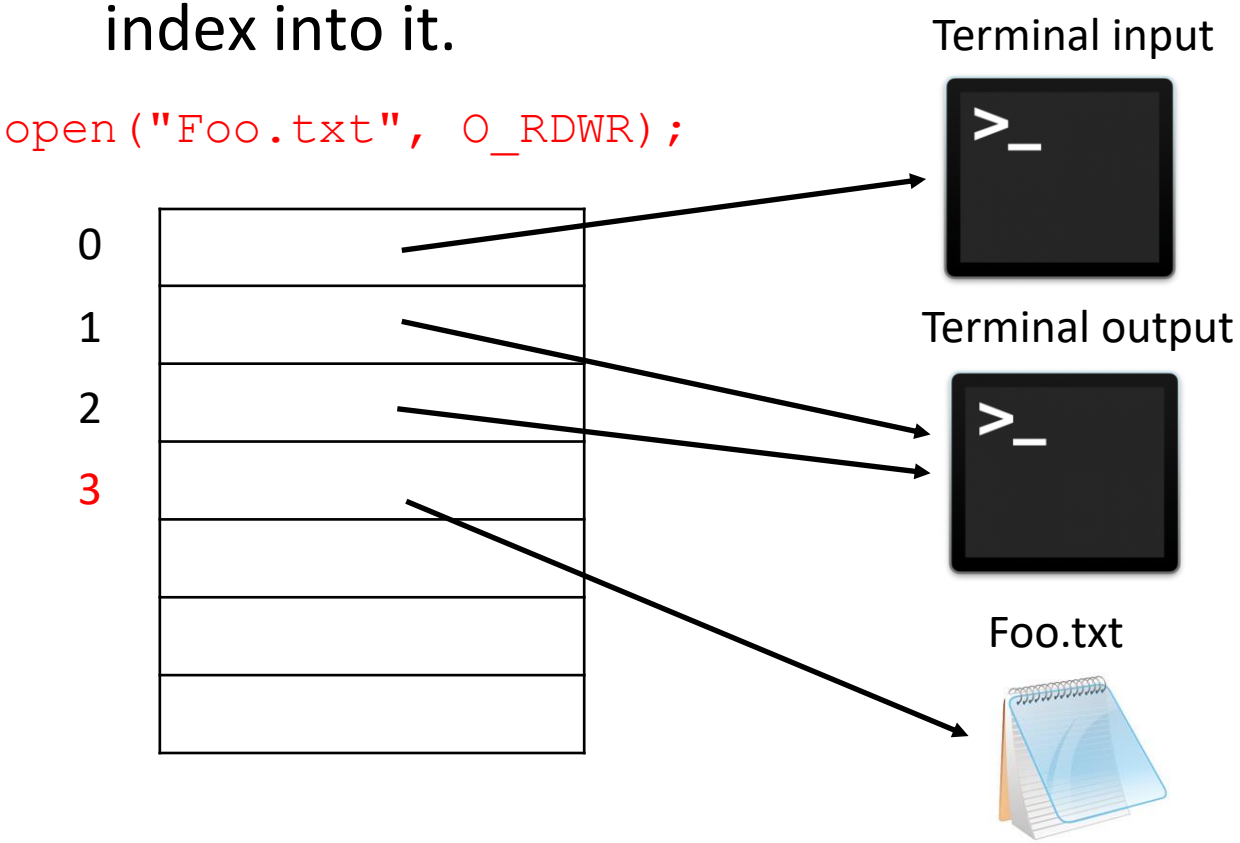
Lecture Outline

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

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);
```

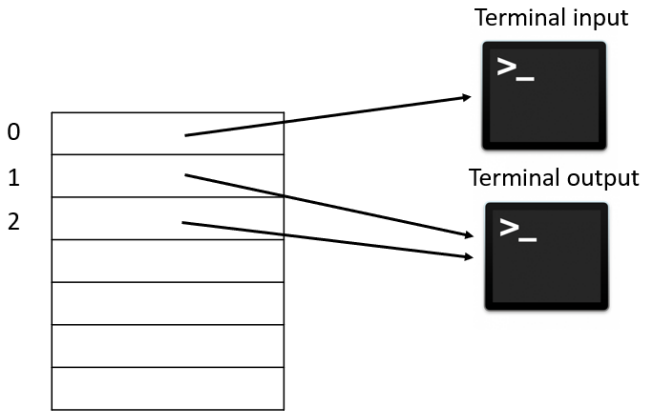


pollev.com/tqm

- What if there was only one global file descriptor table? What negative affects may this have?

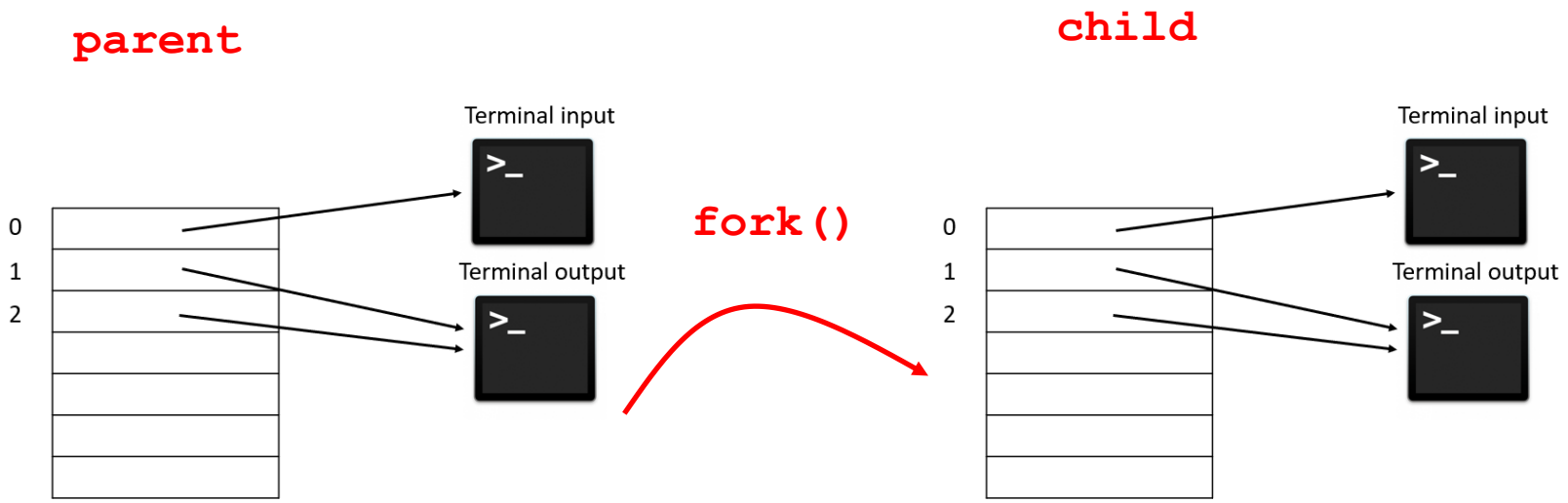
File Descriptor Table: Per Process

- 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



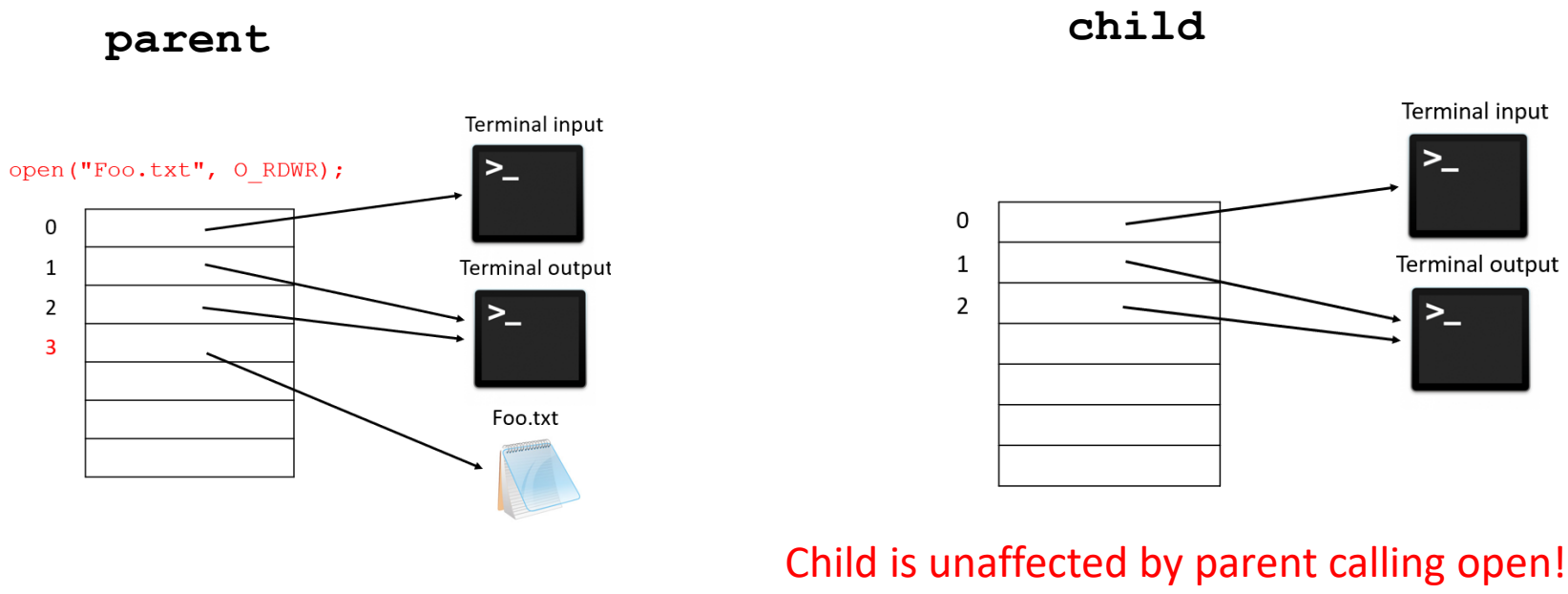
File Descriptor Table: Per Process

- 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

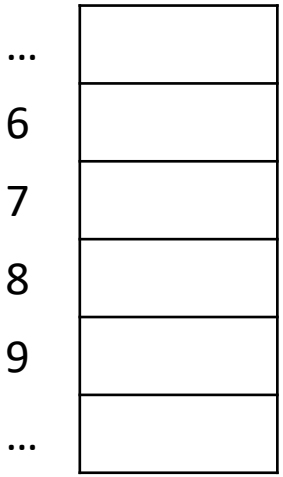


Lecture outline

- Intro to file descriptors
- **File Descriptors: Big picture**
  - **File Descriptors: THE BIGGER PICTURE**
- Redirection & Pipes
- Unix Commands & Controls

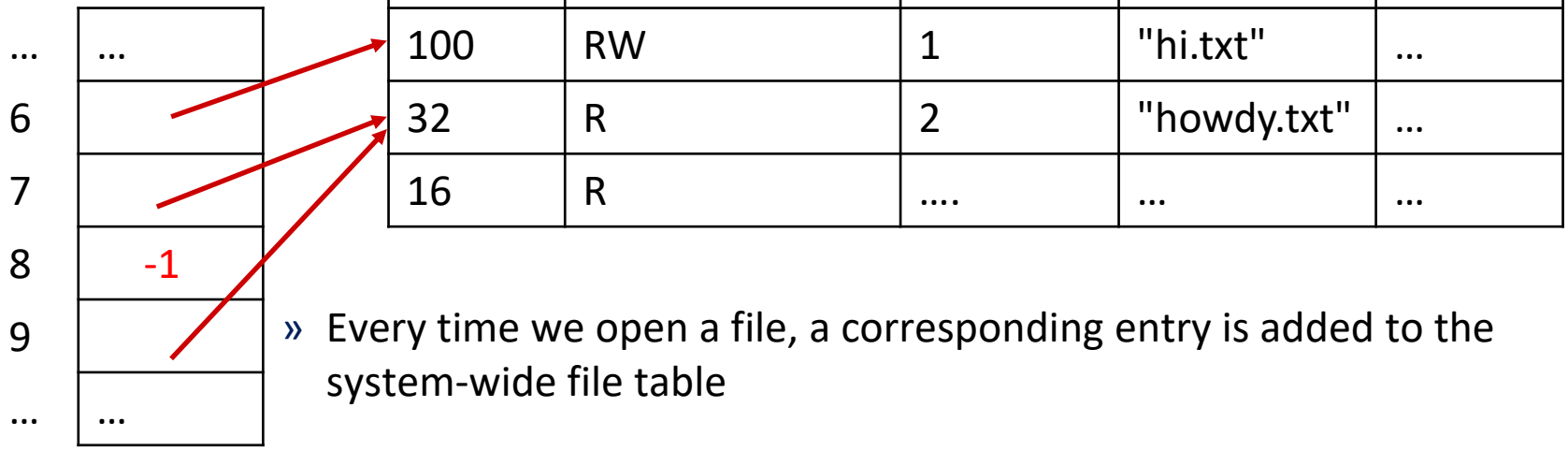
File Descriptor Table in more depth

- Each Process has its own file descriptor table
  - We index into the file descriptor table using the file descriptor



File Descriptor Table in more depth

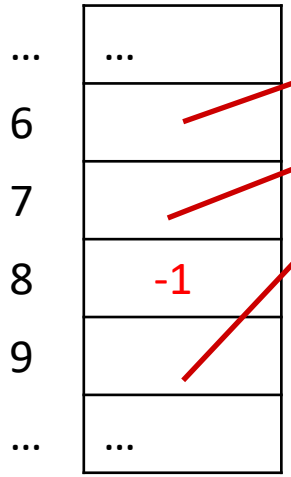
- Each Process has its own file descriptor table
  - We index into the file descriptor table using the file descriptor
  - Each entry in a process's file descriptor table refers to a system-wide file table



» Every time we open a file, a corresponding entry is added to the system-wide file table

File Descriptor Table in more depth

- Each Process has its own file descriptor table
  - We index into the file descriptor table using the file descriptor
  - Each entry in a process's file descriptor table refers to a system-wide file table

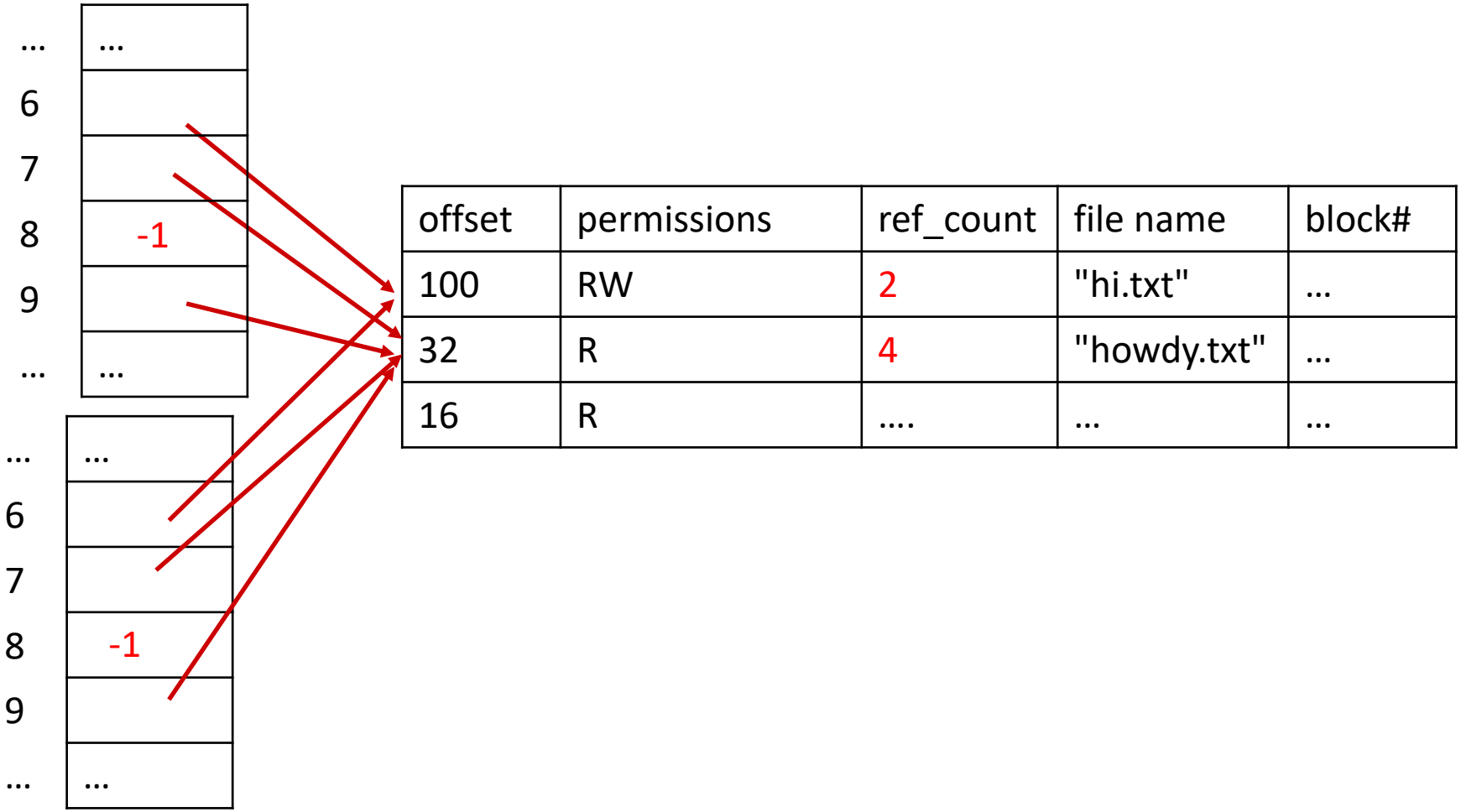


offset	permissions	ref_count	file name	block#
100	RW	1	"hi.txt"	…
32	R	2	"howdy.txt"	…
16	R	….	…	…

- » Every time we open a file, a corresponding entry is added to the system-wide file table
- » Multiple file descriptors can refer to the same system-wide file table entry (see the dup() and dup2() commands)

File Descriptor Table in more depth

- What if we fork? The child copies the parent's file descriptor table



File Descriptor Table in more depth

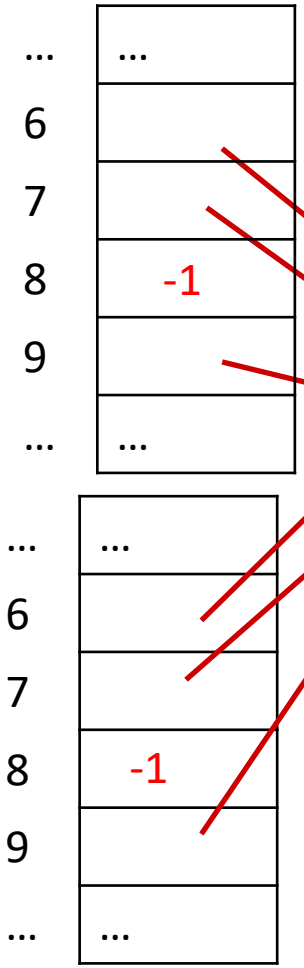
- What if we fork? The child copies the parent's file descriptor table

offset	permissions	ref_count	file name	block#
100	RW	2	"hi.txt"	…
32	R	4	"howdy.txt"	…
16	R	….	…	…

Offset is the current offset into the file. So the next time someone reads/writes "hi.txt", it will start at position 100 into the file

File Descriptor Table in more depth

- What if we fork? The child copies the parent's file descriptor table

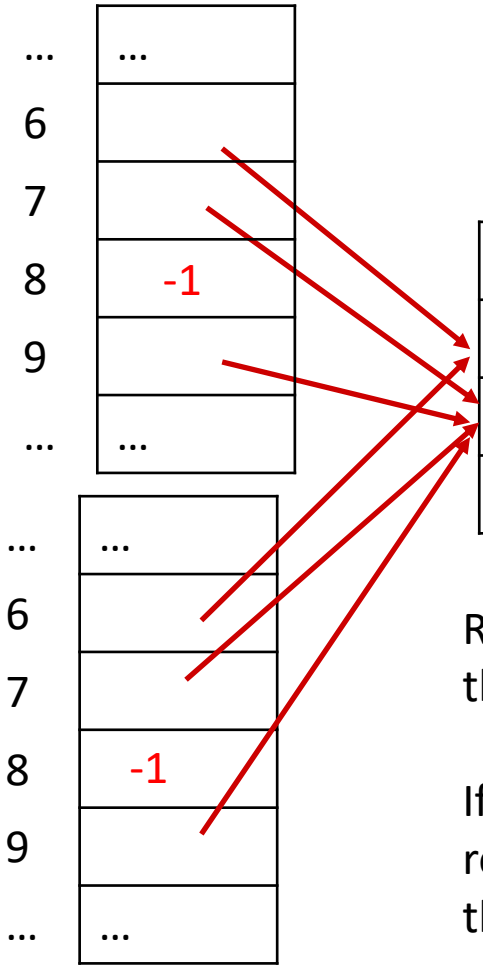


offset	permissions	ref_count	file name	block#
100	RW	2	"hi.txt"	…
32	R	4	"howdy.txt"	…
16	R	….	…	…

Permissions dictate what modes the file was opened with
O_RDONLY (read only)
O_WRONLY (write only)
Or both (or some others I am not covering here)

File Descriptor Table in more depth

- What if we fork? The child copies the parent's file descriptor table



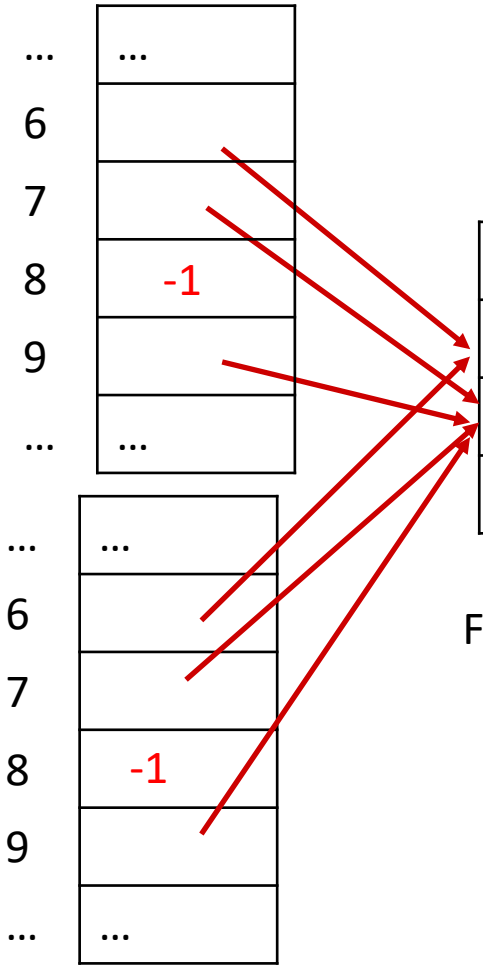
offset	permissions	ref_count	file name	block#
100	RW	2	"hi.txt"	…
32	R	4	"howdy.txt"	…
16	R	….	…	…

Ref count is the number of file descriptors currently referring to that entry in the system-wide table.

If a file descriptor closes, then decrement the ref_count by 1. If the ref_count reaches zero, everyone is done with it and we can close the file in the system-wide table

File Descriptor Table in more depth

- What if we fork? The child copies the parent's file descriptor table

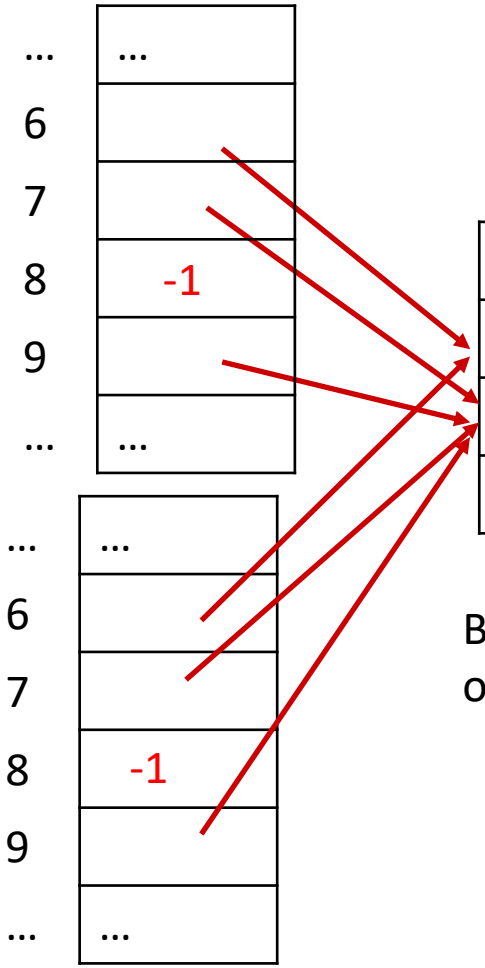


offset	permissions	ref_count	file name	block#
100	RW	2	"hi.txt"	…
32	R	4	"howdy.txt"	…
16	R	….	…	…

Filename is the file name

File Descriptor Table in more depth

- What if we fork? The child copies the parent's file descriptor table

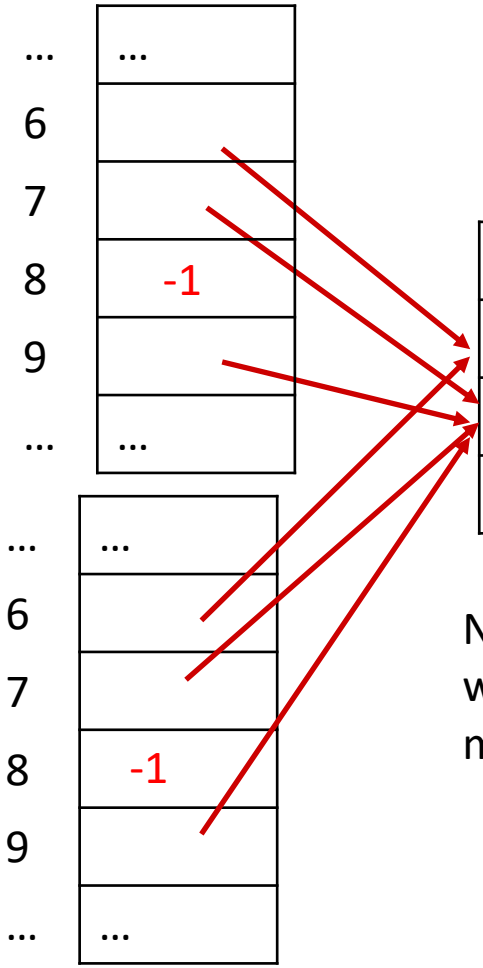


offset	permissions	ref_count	file name	block#
100	RW	2	"hi.txt"	…
32	R	4	"howdy.txt"	…
16	R	….	…	…

Block # is referring to where the file starts in the file system (more on the file system later)

File Descriptor Table in more depth

- What if we fork? The child copies the parent's file descriptor table



offset	permissions	ref_count	file name	block#
100	RW	2	"hi.txt"	…
32	R	4	"howdy.txt"	…
16	R	….	…	…

NOTE: not all file systems are the same, but this idea of a system-wide table and per-process file descriptor tables holds (and you may want to take inspiration for PennOS

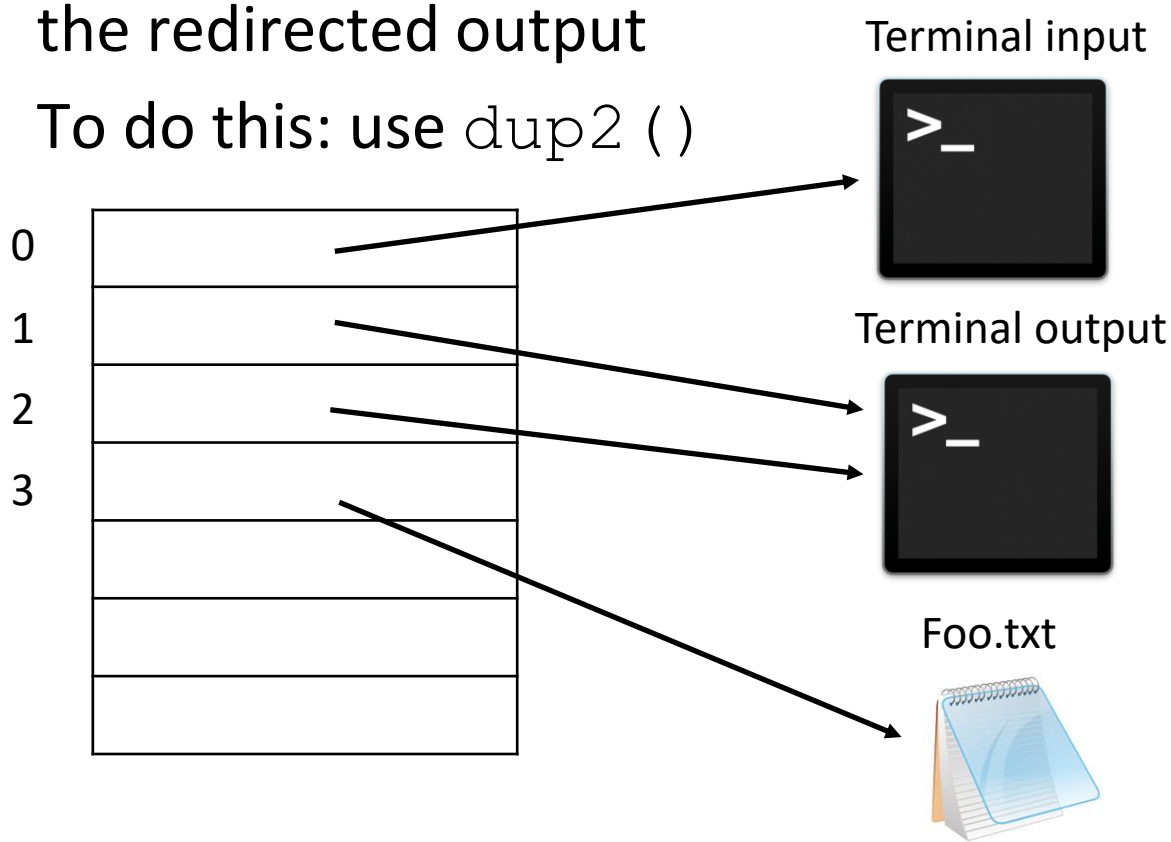
Lecture Outline

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

Redirecting stdin/out/err

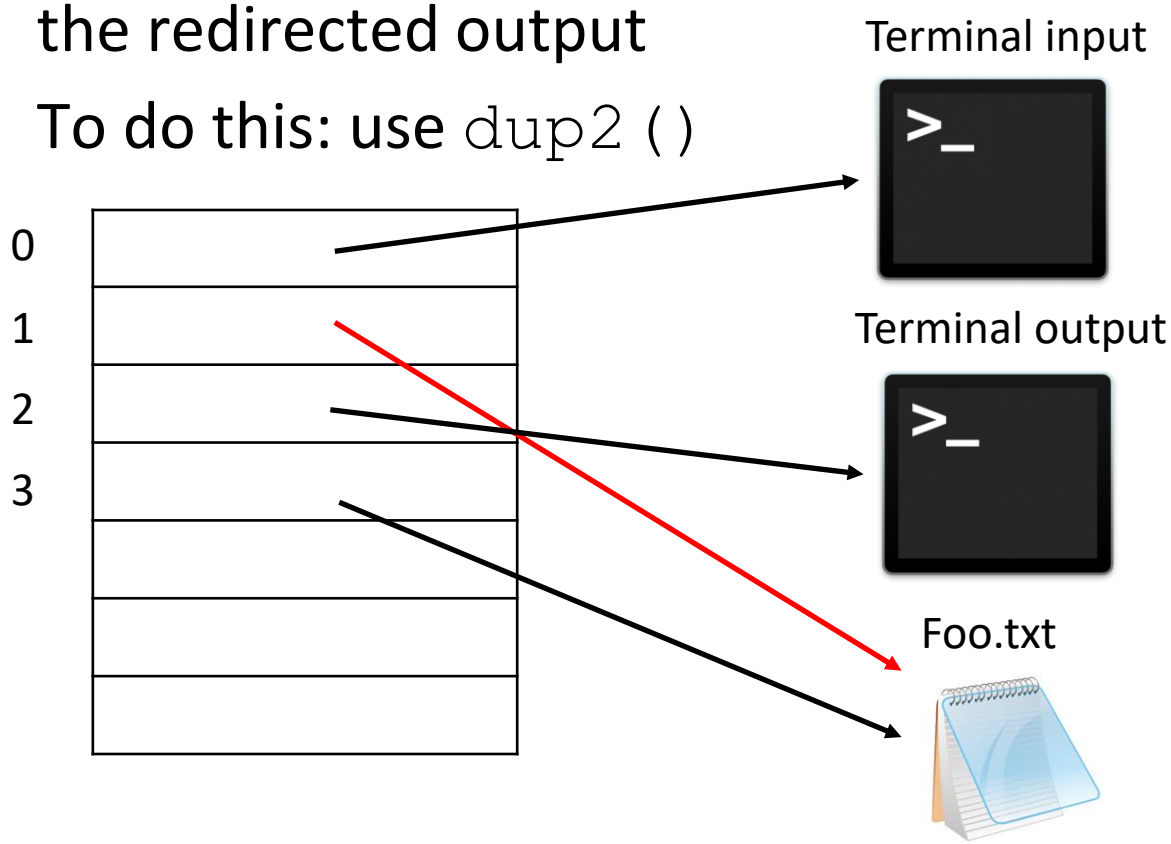
`printf` is implemented using `write(STDOUT_FILENO`
That's why it is redirected after changing stdout

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



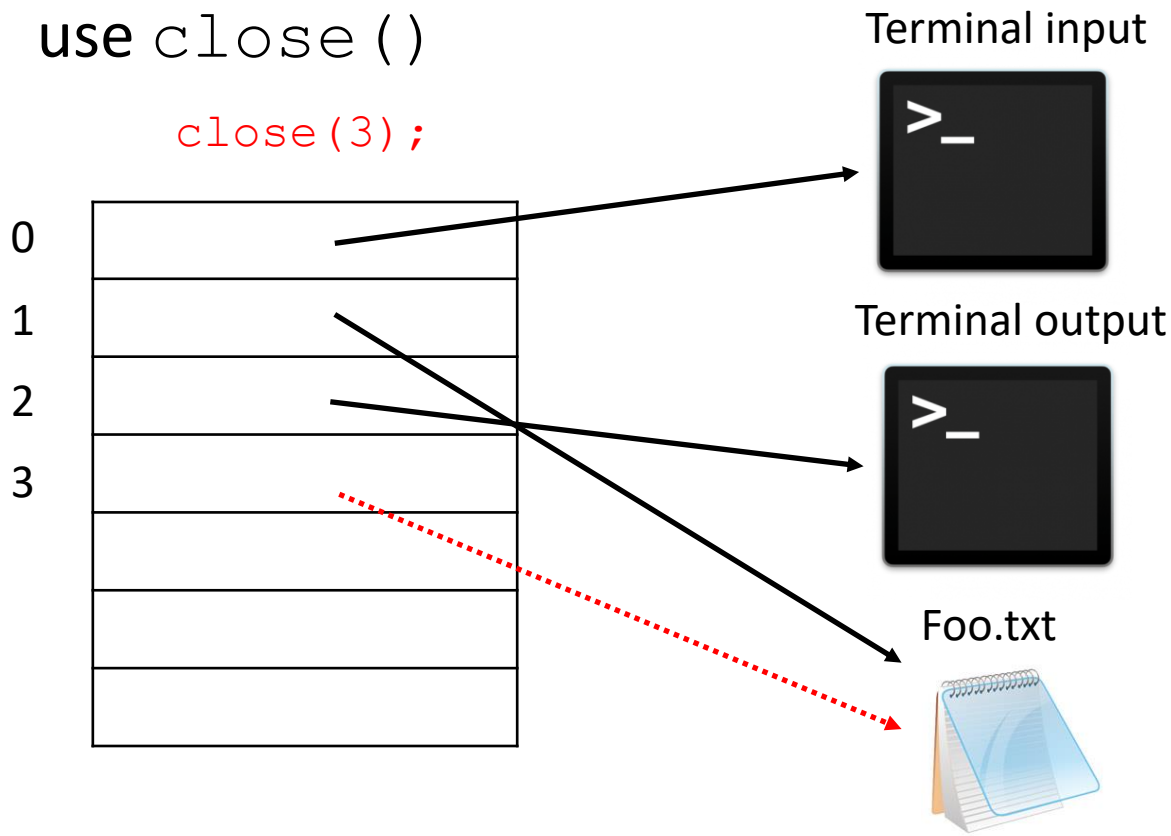
Redirecting stdin/out/err

- We can change things so that STDOUT_FILENO is associated with something other than a terminal output.
- Now, any calls to printf, stdout, 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

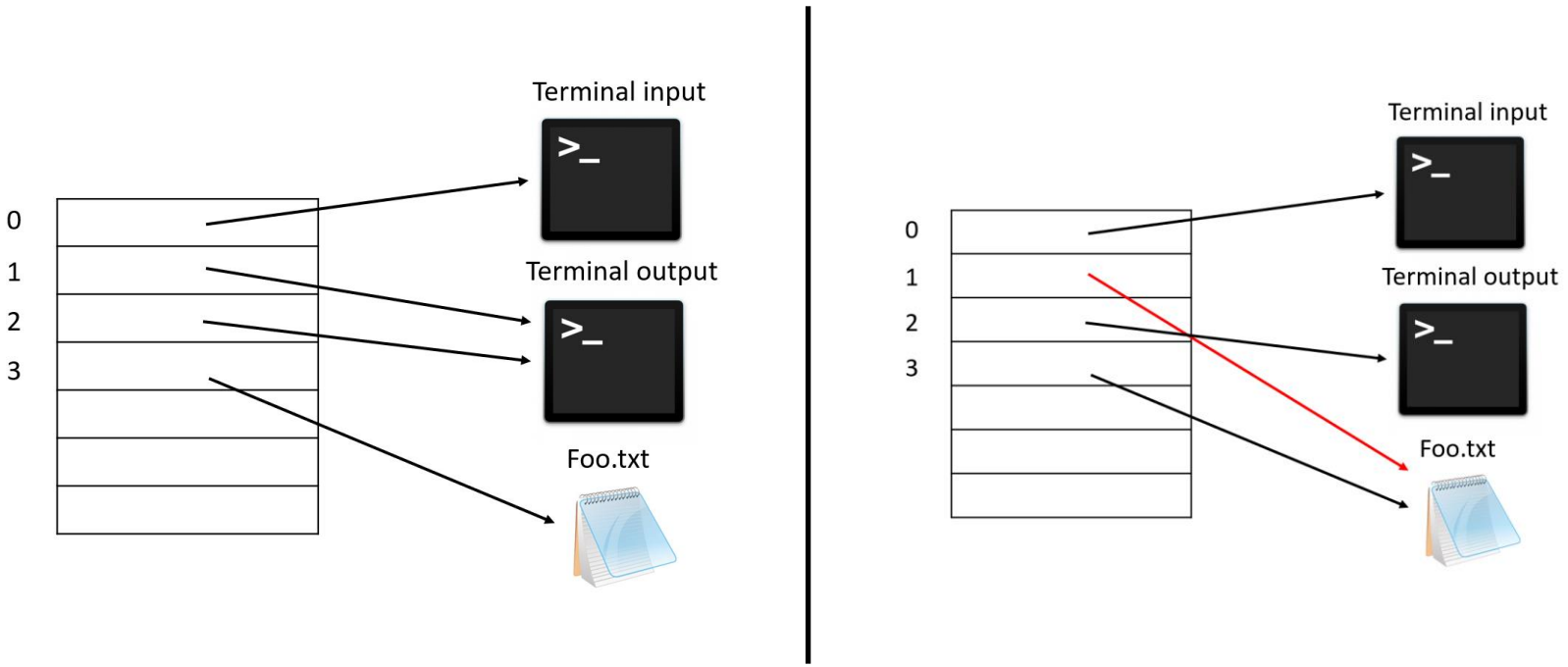
- ```
  int dup2(int oldfd, int newfd);
  ```

- Look it up in the man pages ☺
  ```

Poll Everywhere

pollev.com/tqm

- Based on the man page for dup2, what do code do you have to write to achieve this redirection?



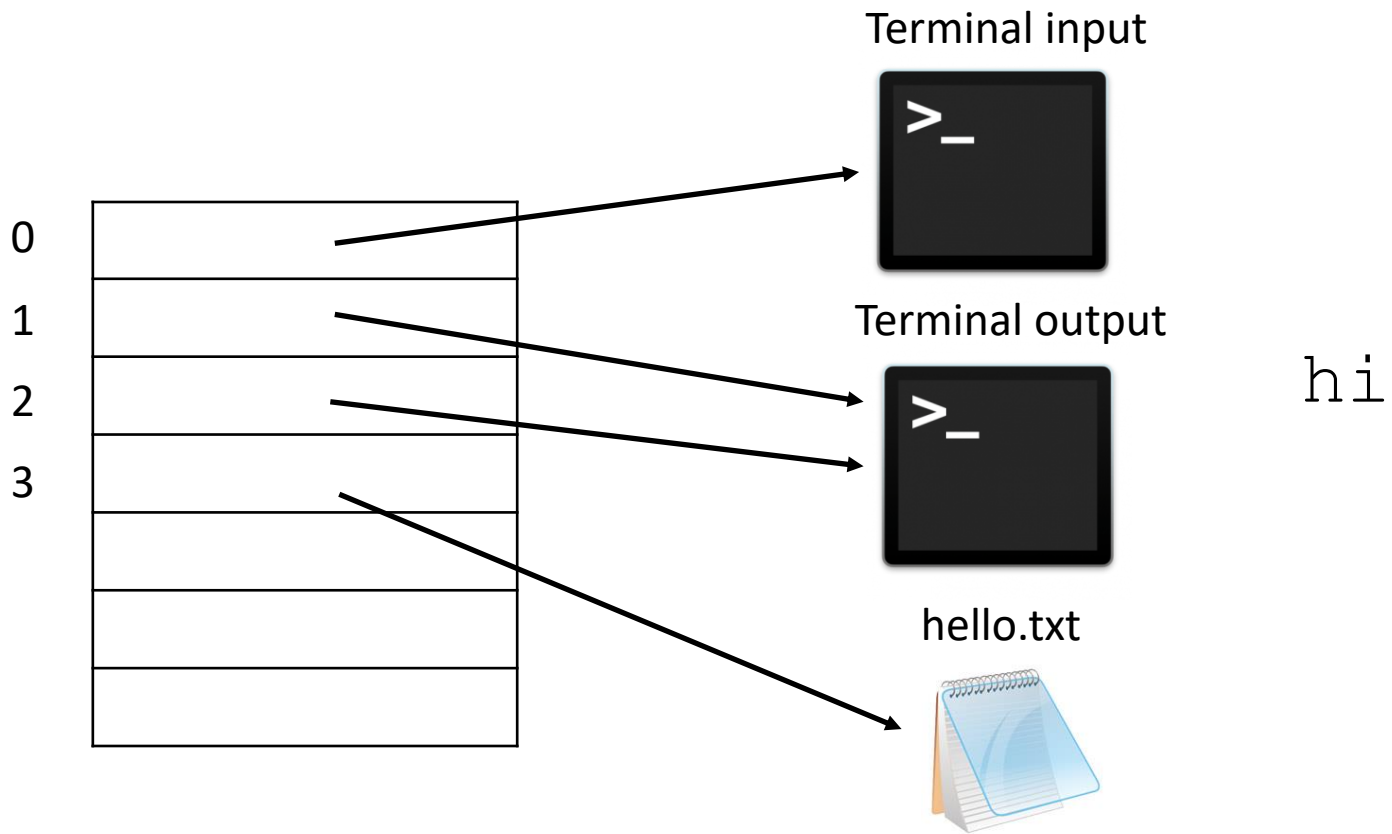
pollev.com/tqm

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

```c
int main() {
  int fd = open("hello.txt", O_WRONLY);

  printf("hi\n");

  close(STDOUT_FILENO);

  printf("?\n");

  // open `fd` on `stdout`
  dup2(fd, STDOUT_FILENO);

  printf("!\n");

  close(fd);

  printf("*\n");

}
```

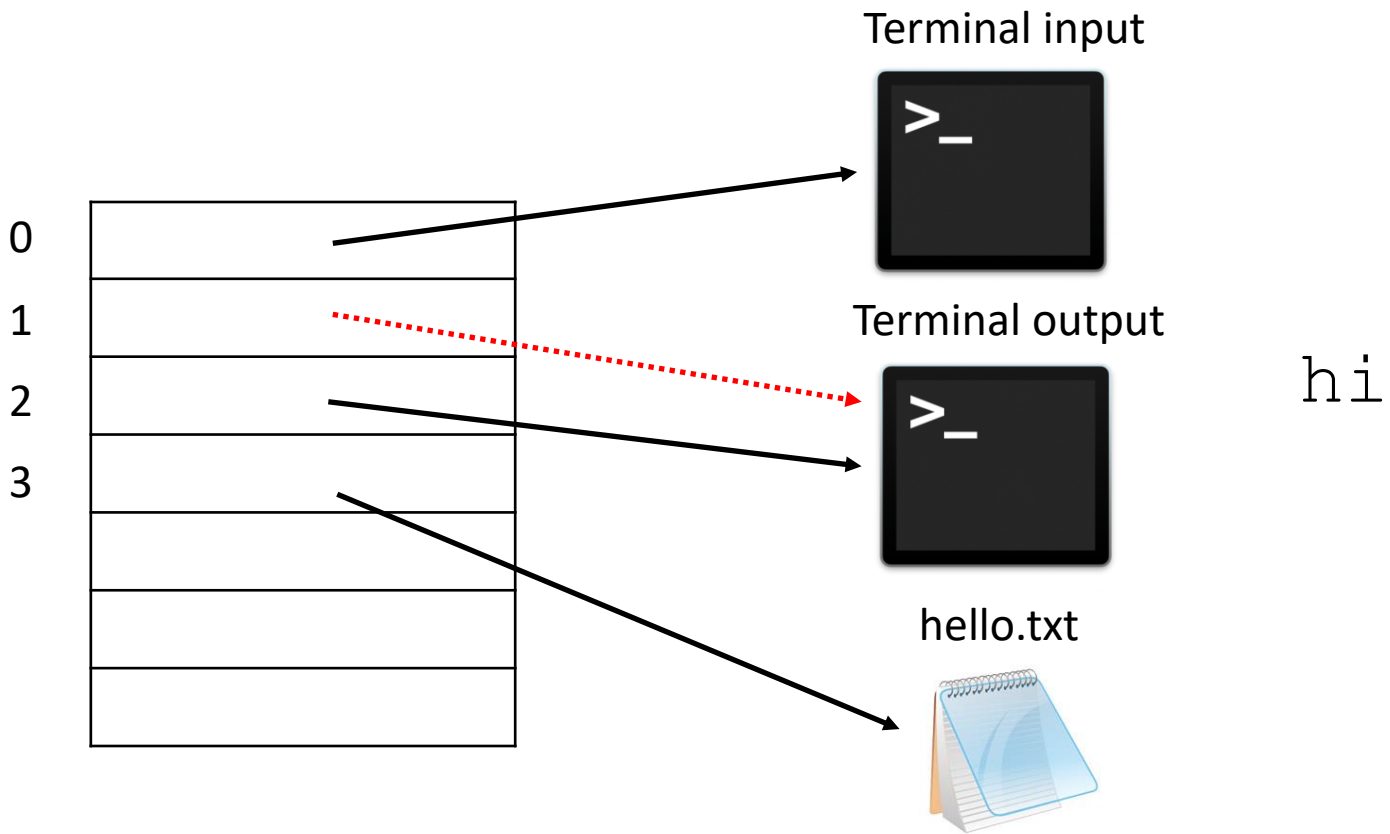
Explanation

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



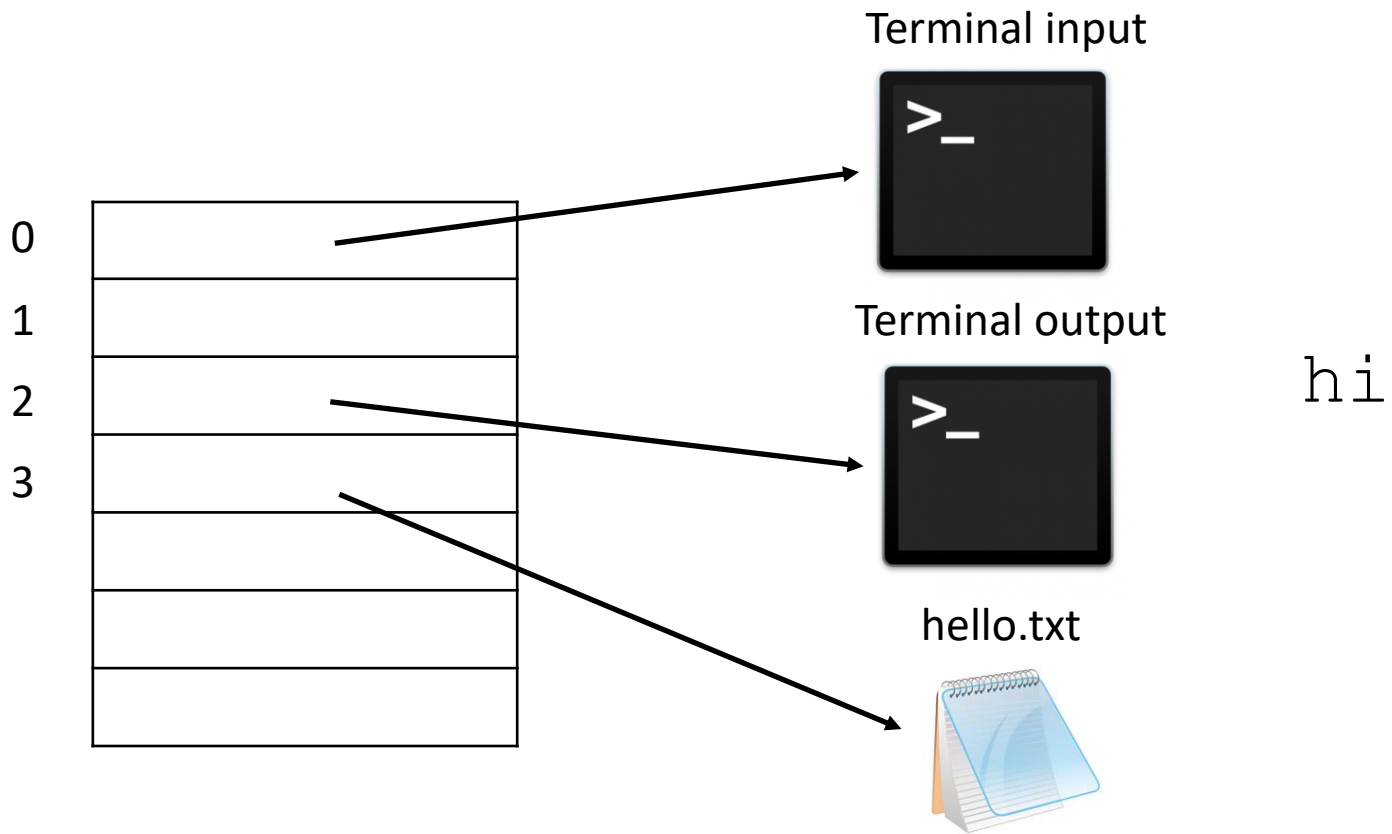
Explanation

```
close(STDOUT_FILENO);
```



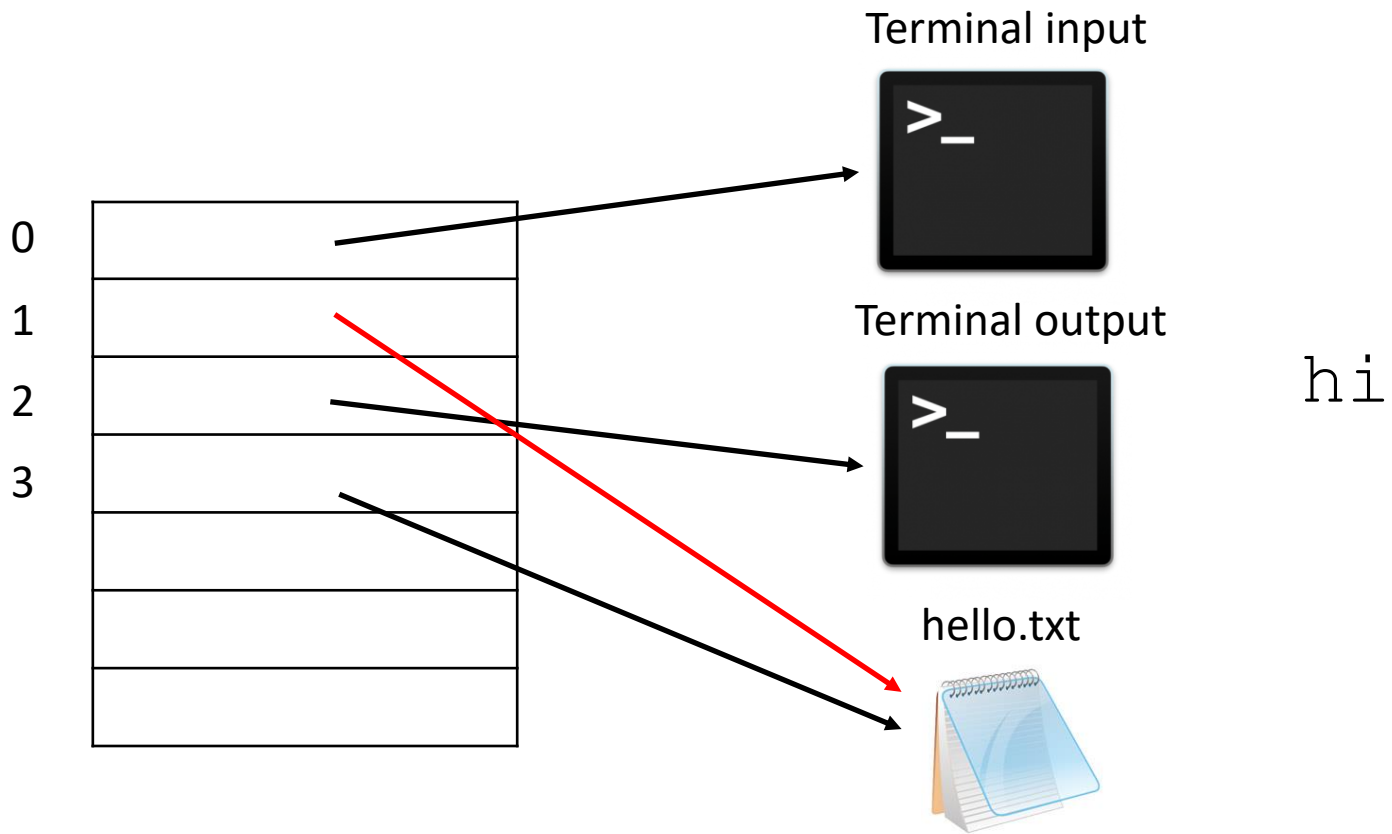
Explanation

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



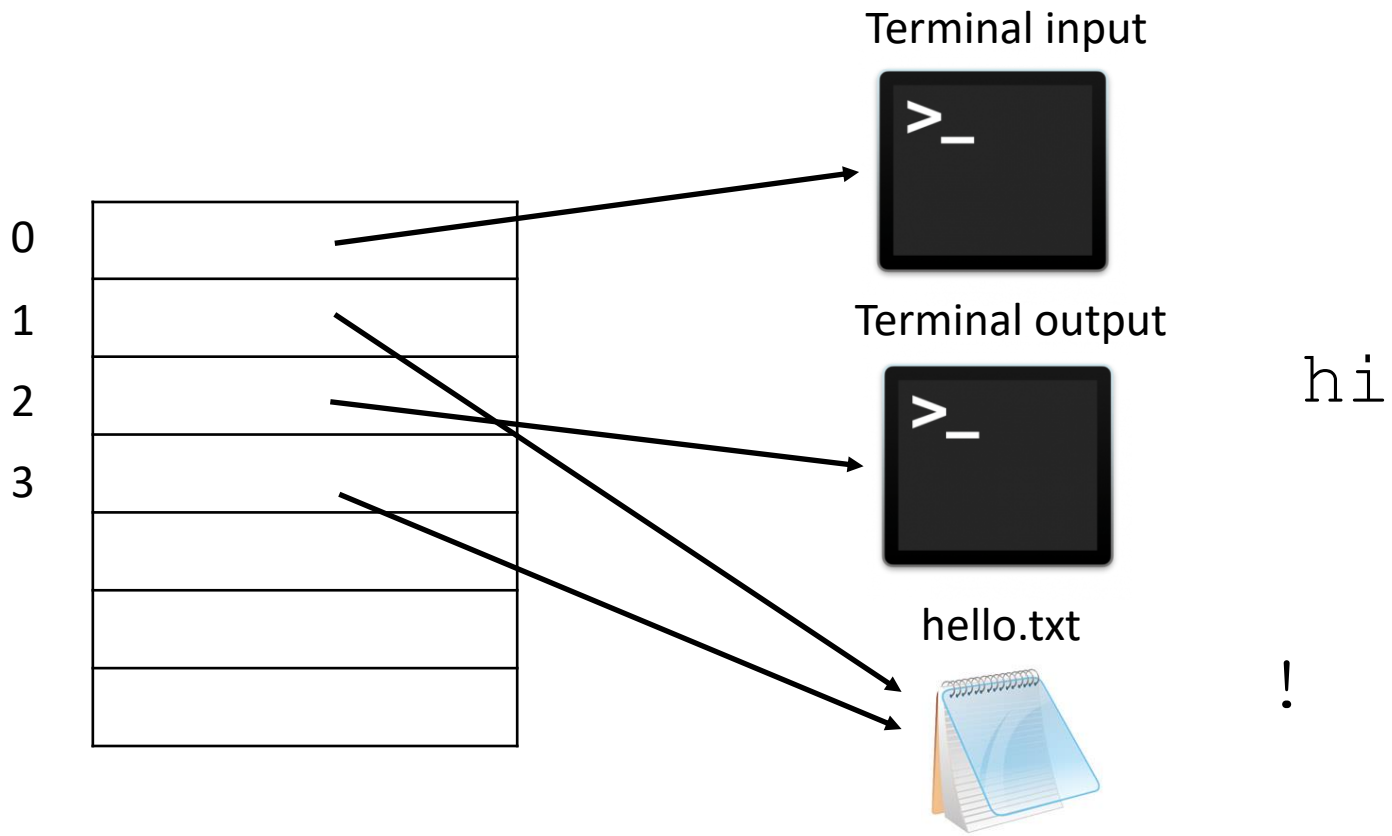
Explanation

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



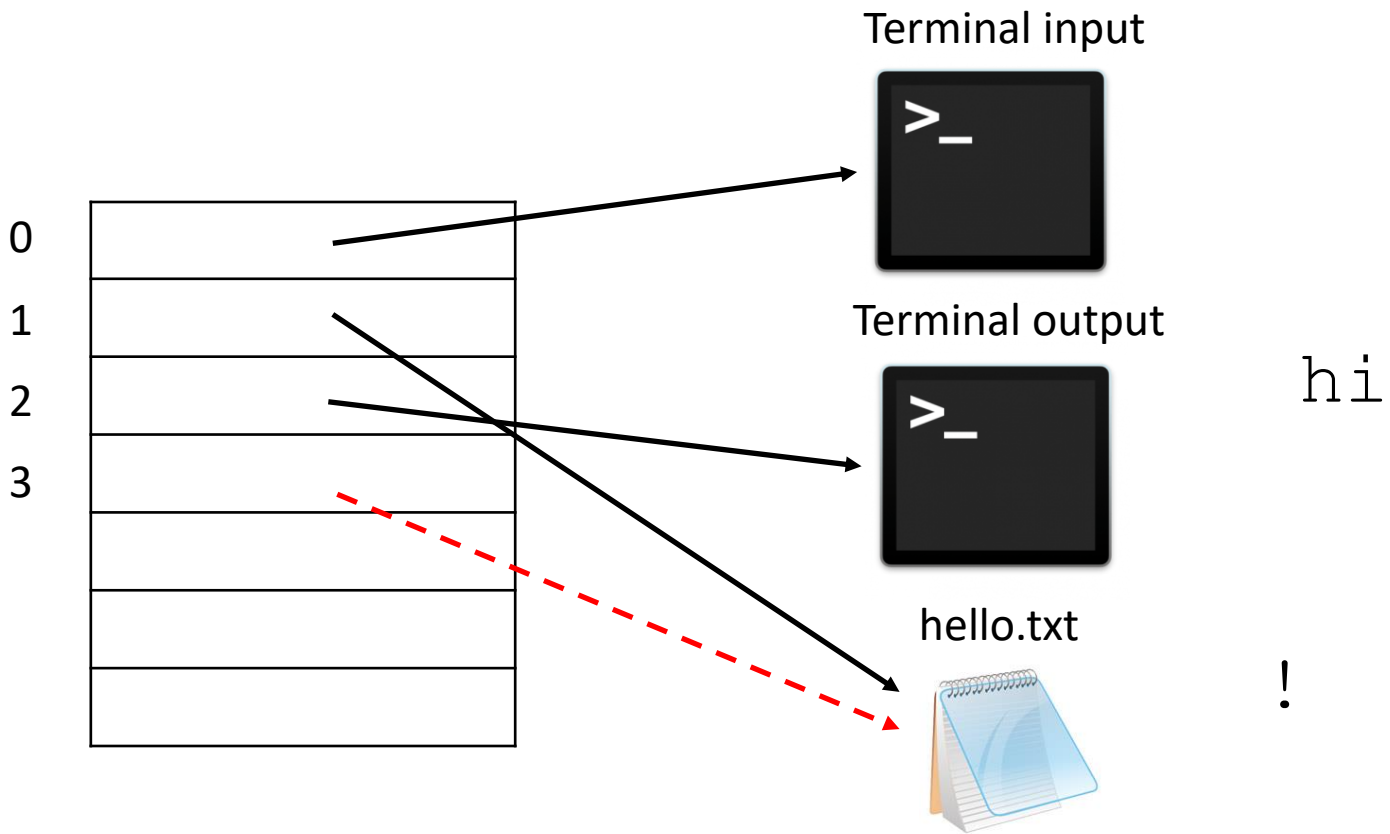
Explanation

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



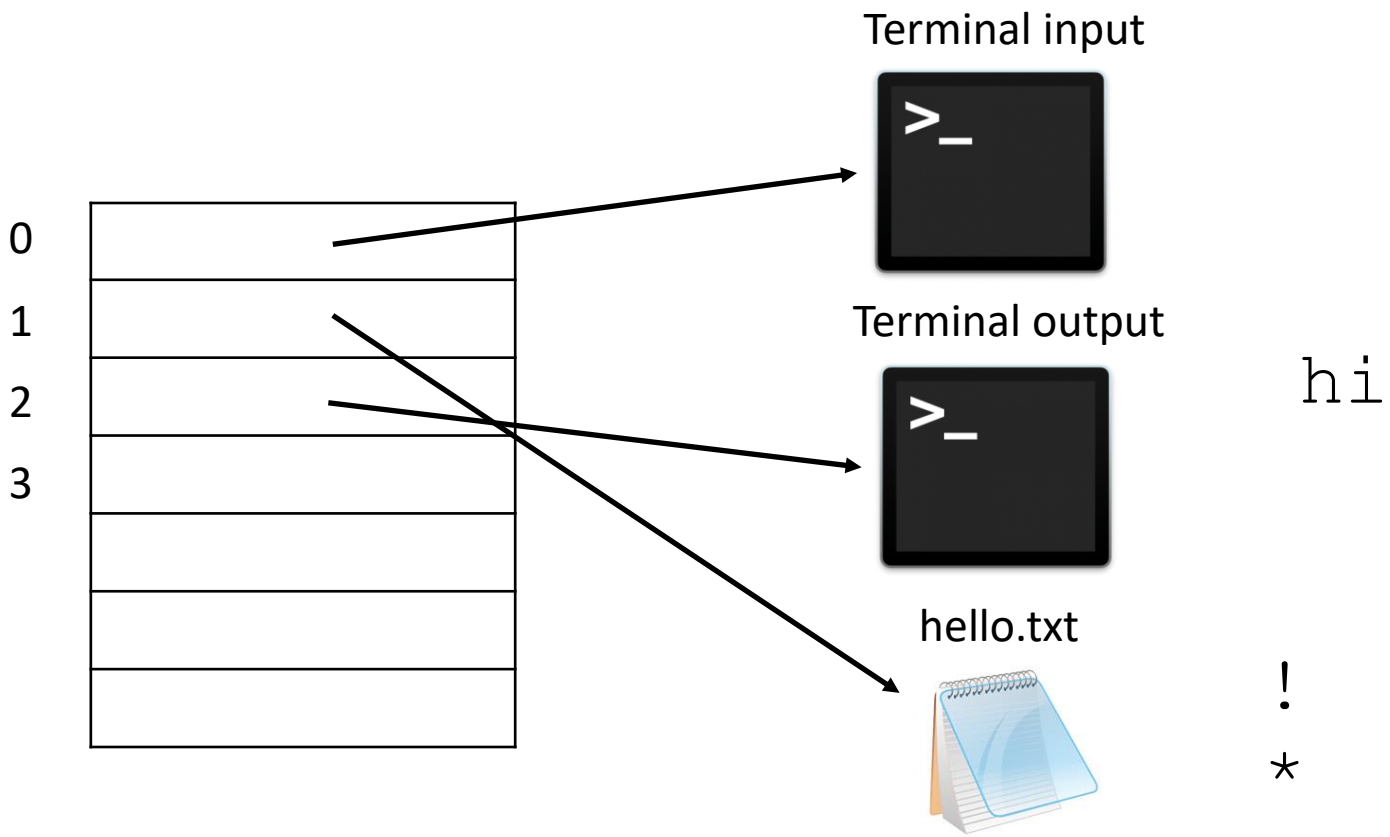
Explanation

```
close(fd);
```



Explanation

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



Pipes

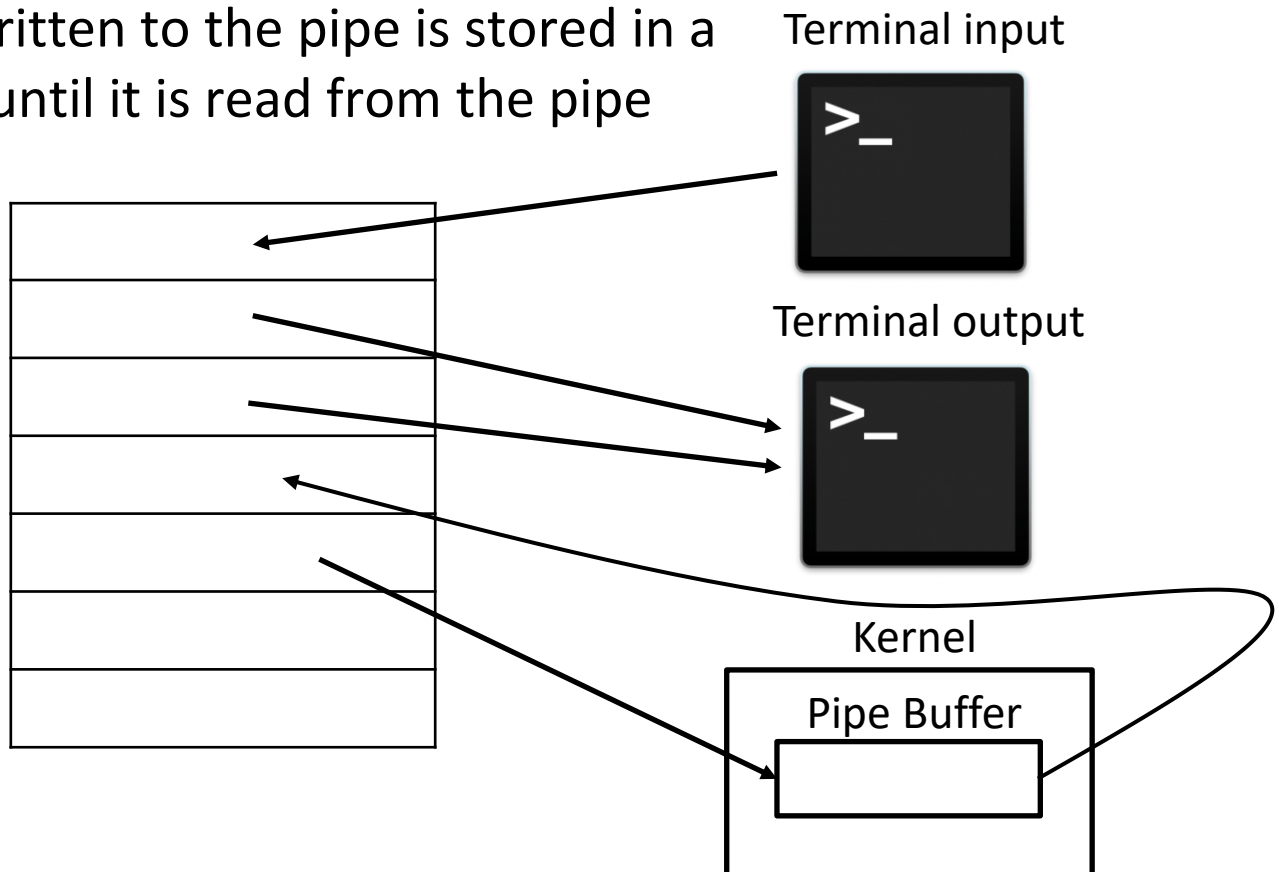
```
int pipe(int pipefd[2]);
```

- 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



pollev.com/tqm

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

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main() {
  int pipe_fds[2];
  pipe(pipe_fds);

  pid_t pid = fork();

  if (pid == 0) {
    /// close my end of the pipe
    close(pipe_fds[0]);

    write(pipe_fds[1], "Hello!", 6);

    char str[7];
    ssize_t chars_read = read(pipe_fds[1], str, 6);

    if (chars_read != -1) {
      str[chars_read] = '\0';

      printf("%s\n", str);
    }

    exit(EXIT_SUCCESS);
  }
  // parent

  // parent

  /// close my end of the pipe
  close(pipe_fds[1]);

  char str[7];
  ssize_t chars_read = read(pipe_fds[0], str, 6);

  if (chars_read != -1) {
    str[chars_read] = '\0';
    printf("%s\n", str);
  }

  write(pipe_fds[0], "Howdy!", 6);

  return EXIT_SUCCESS;
}
```

Pipes & EOF

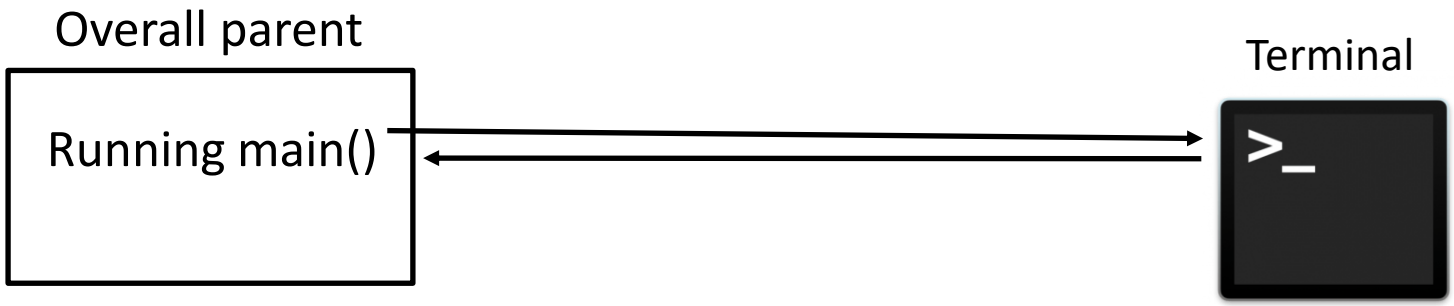
- Many programs will read from a file until they hit EOF and will not terminate until then
- Like reading from the terminal, just because there is nothing in the pipe, does not mean nothing else will ever come through the pipe.
  - EOF is not read in this case
- EOF is only read from a pipe when:
  -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

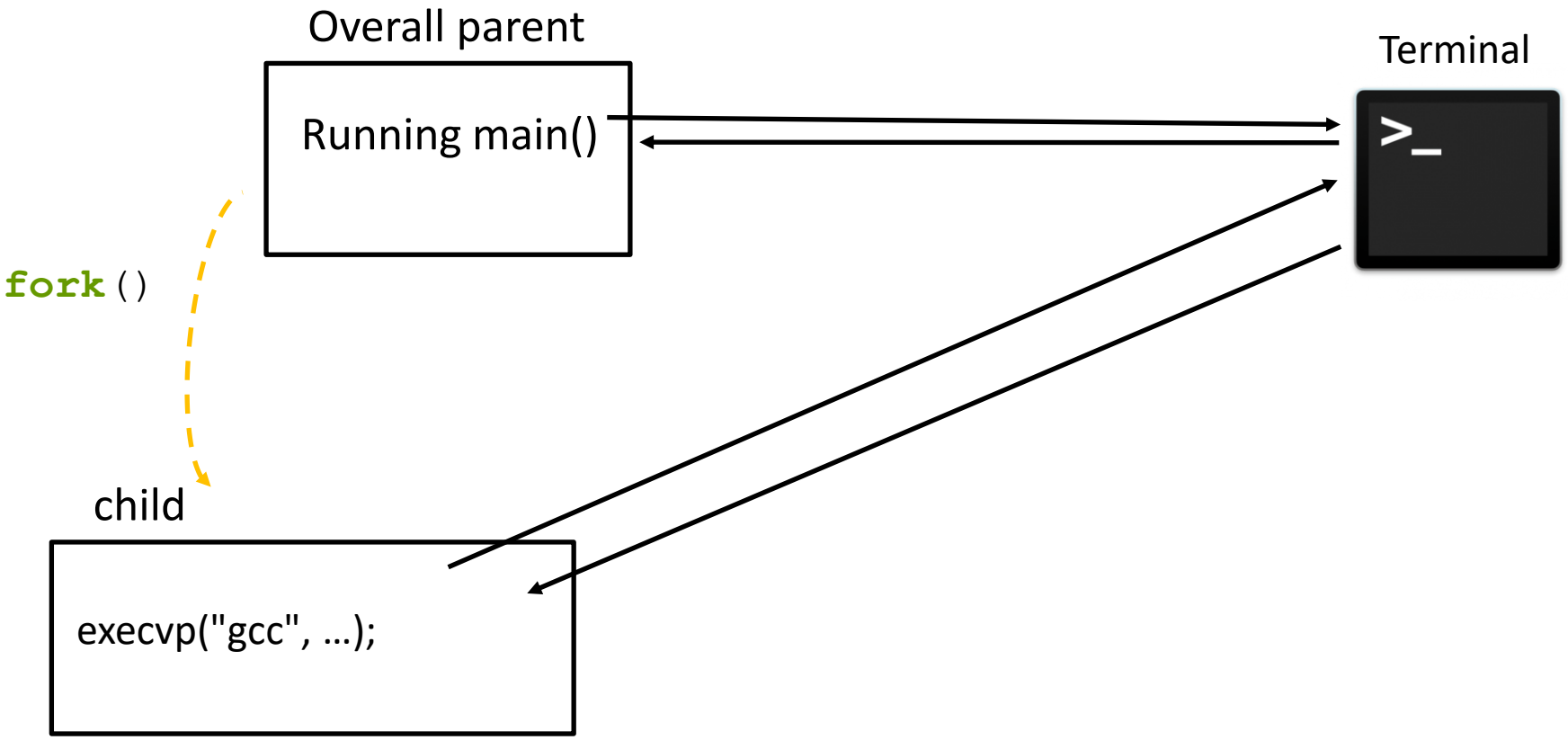
`io_autograder.c` Trace

- First:
  we compile the program with the gcc command



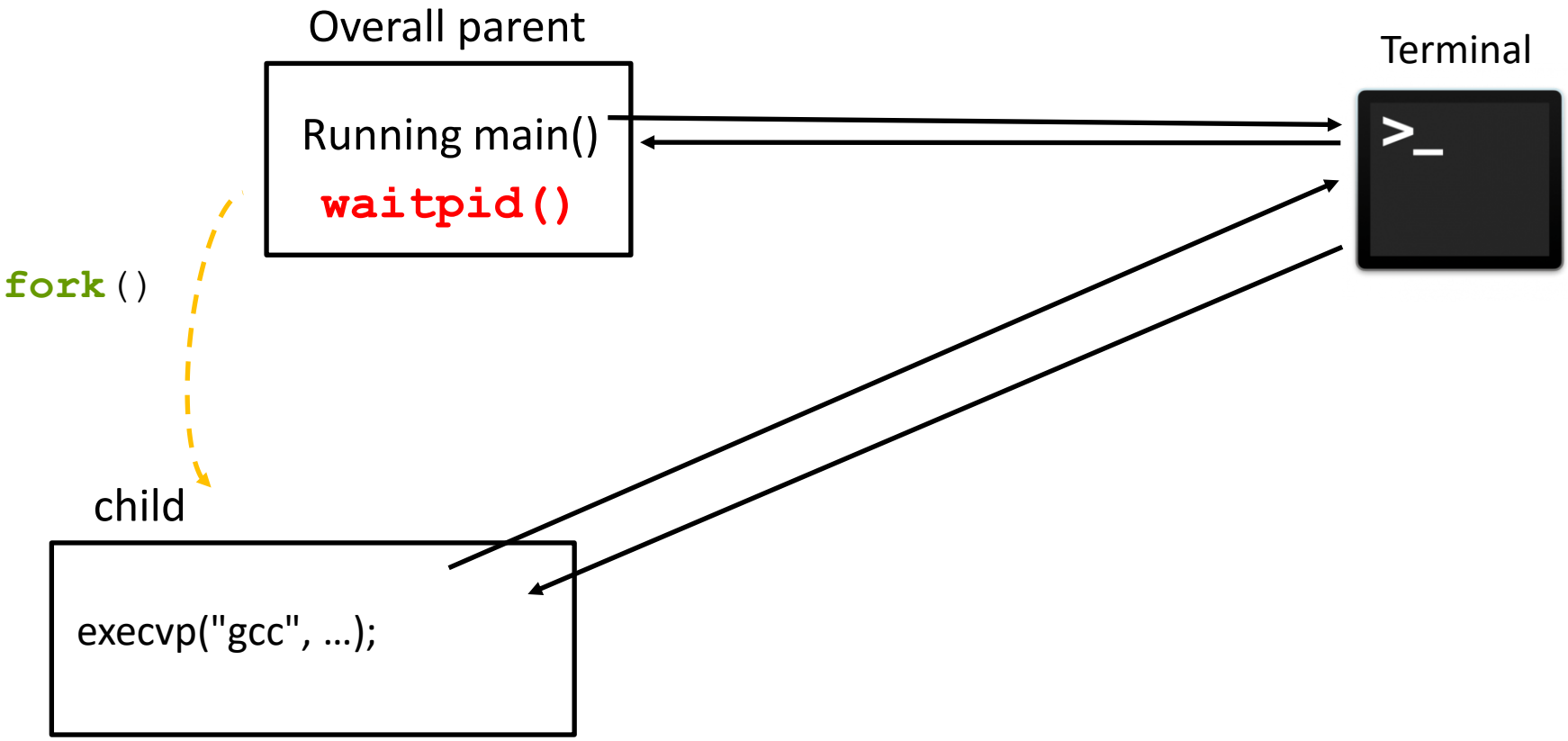
io_autograder.c Trace

- First:
  we compile the program with the gcc command



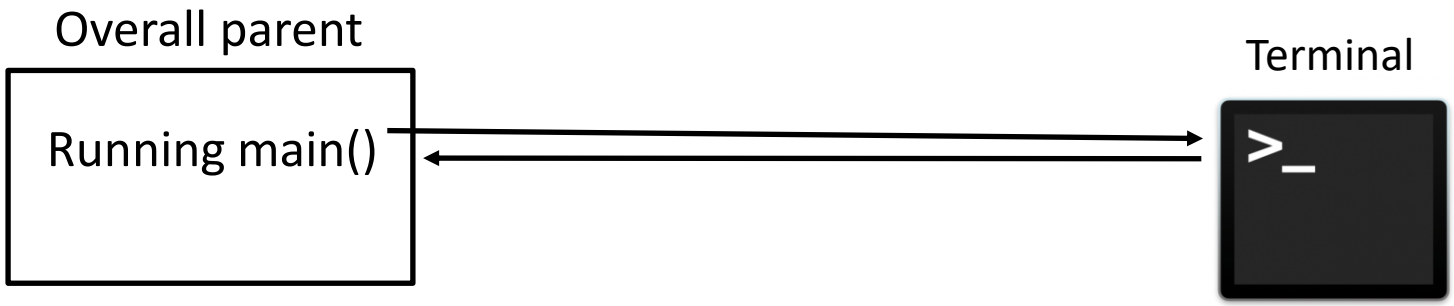
io_autograder.c Trace

- First:
  we compile the program with the gcc command



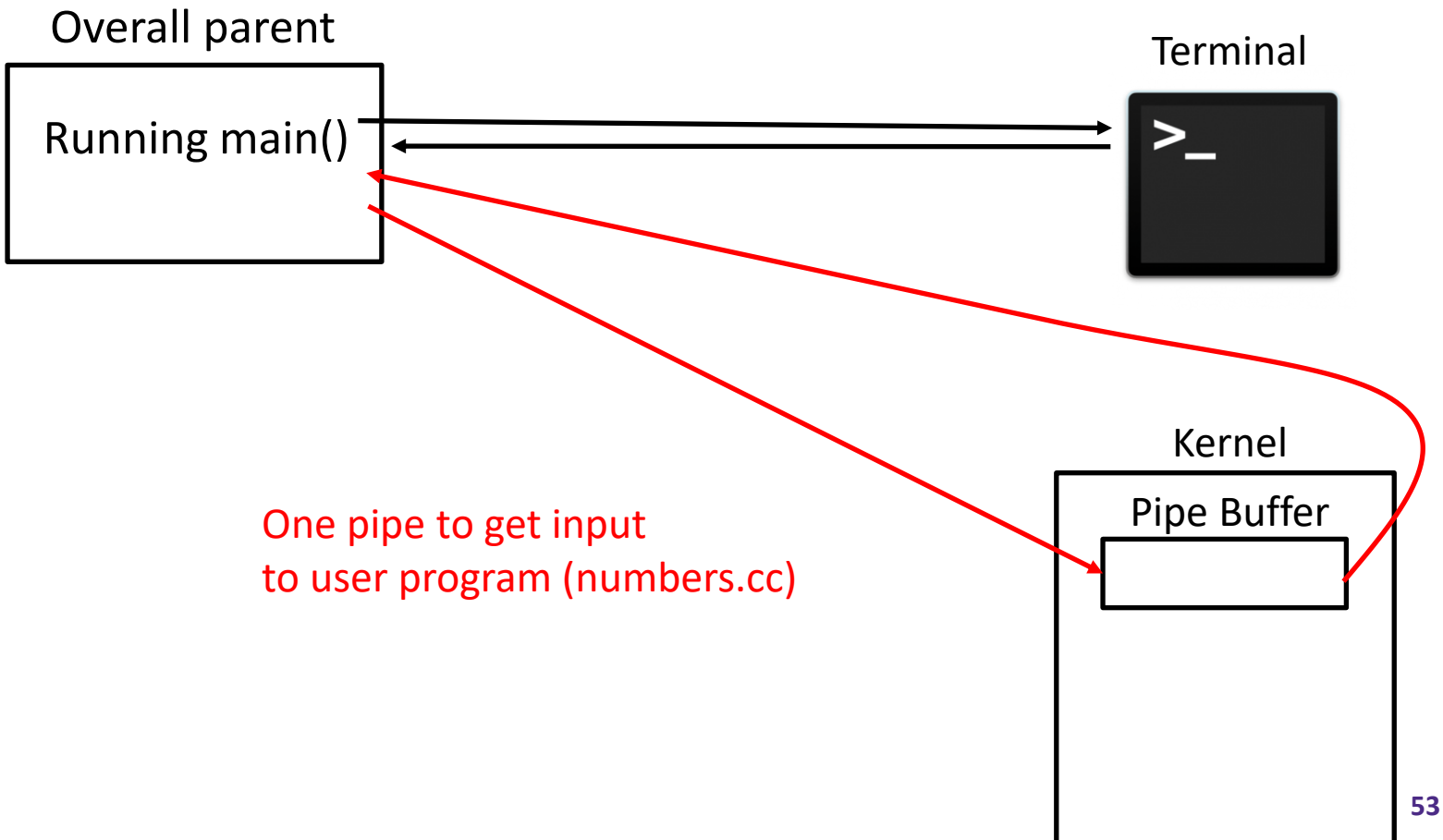
io_autograder.c Trace

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



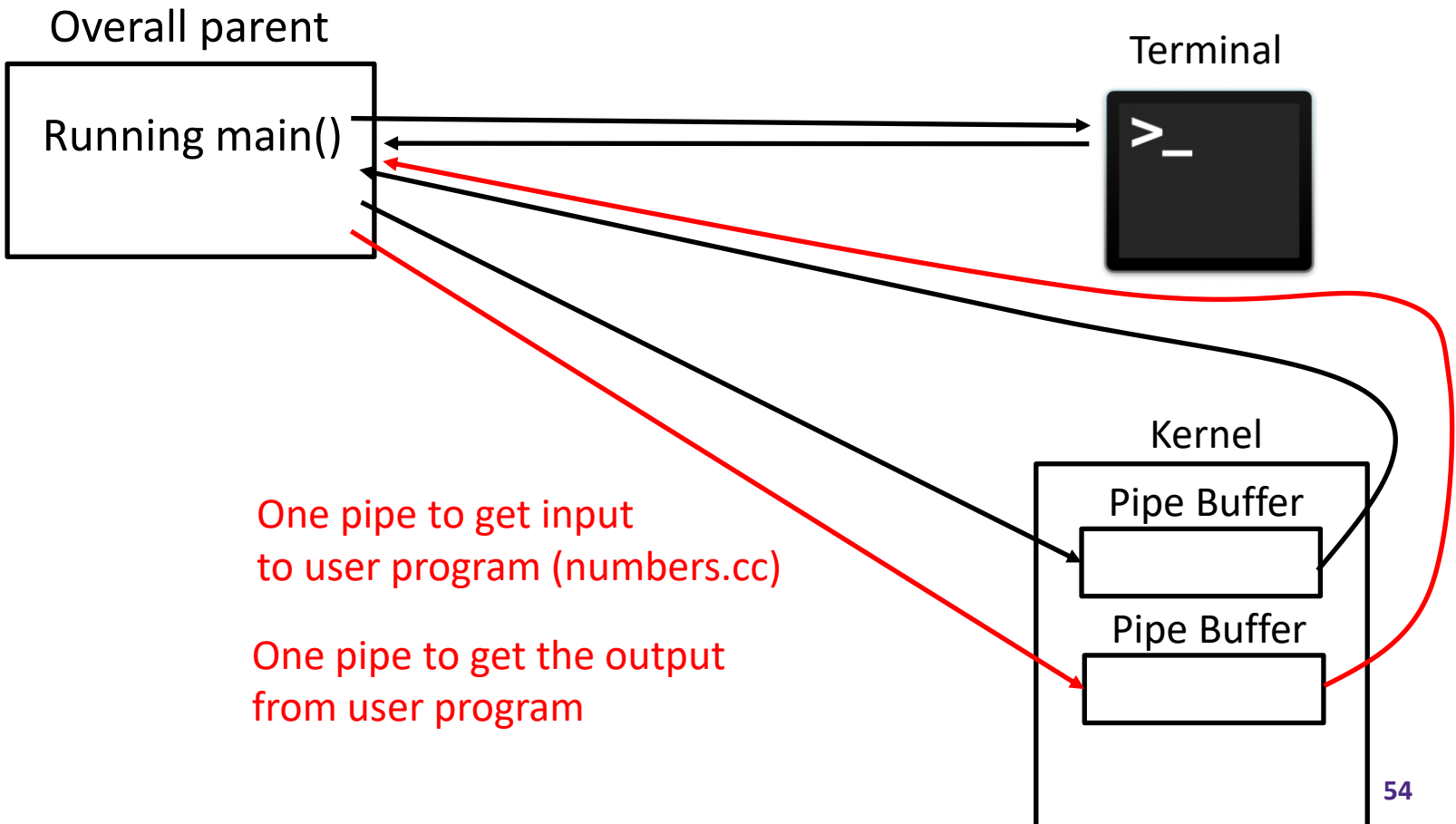
io_autograder.c Trace

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



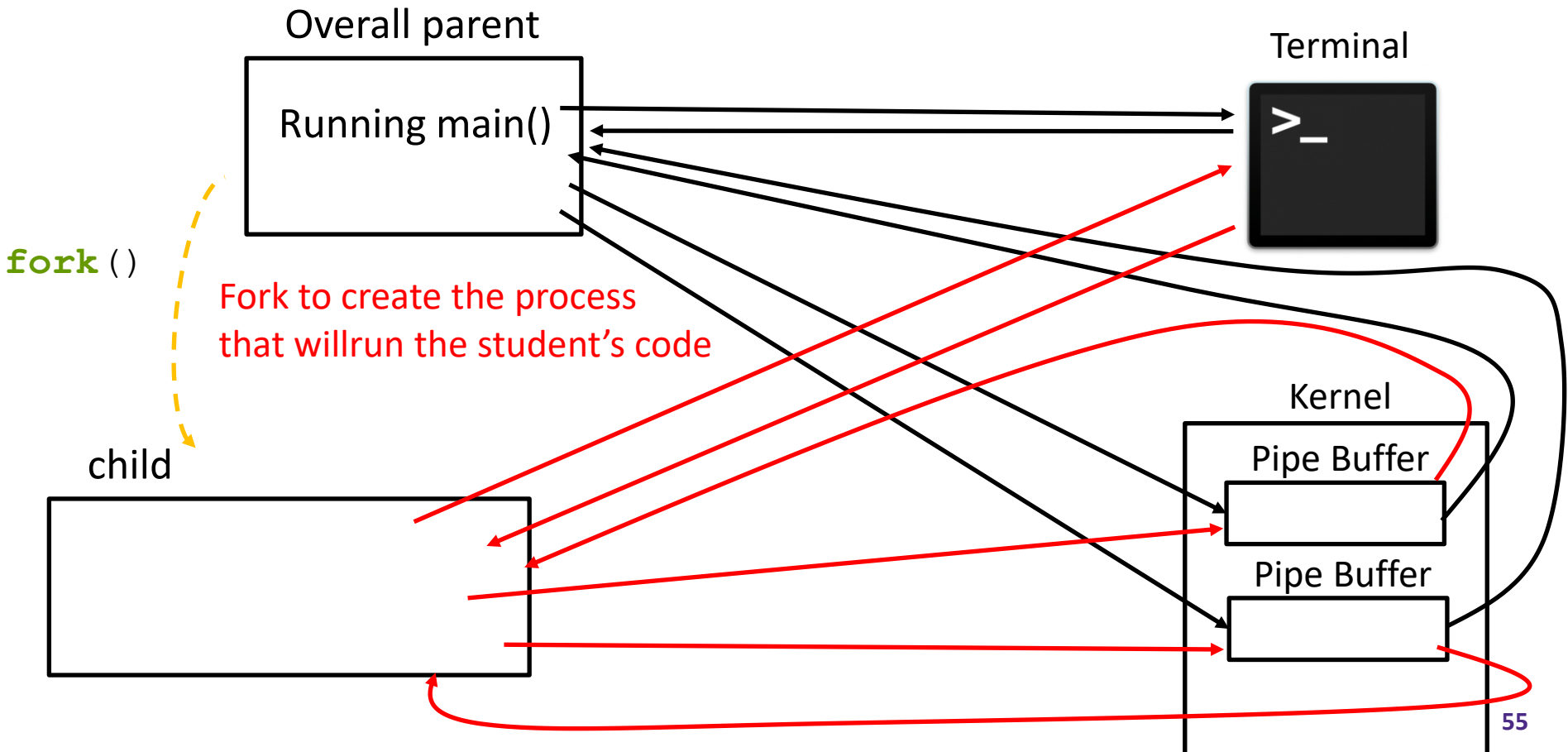
io_autograder.c Trace

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



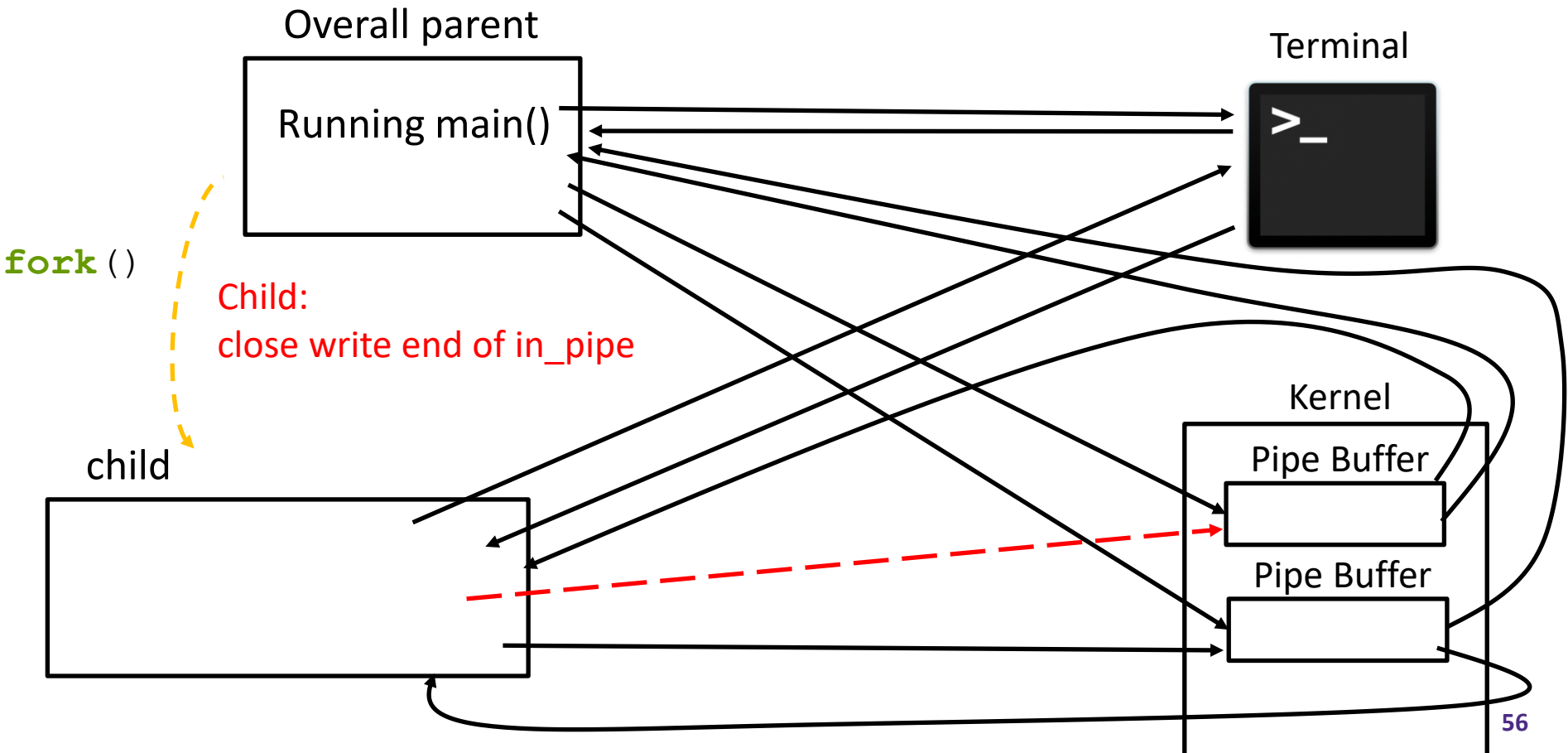
io_autograder.c Trace

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



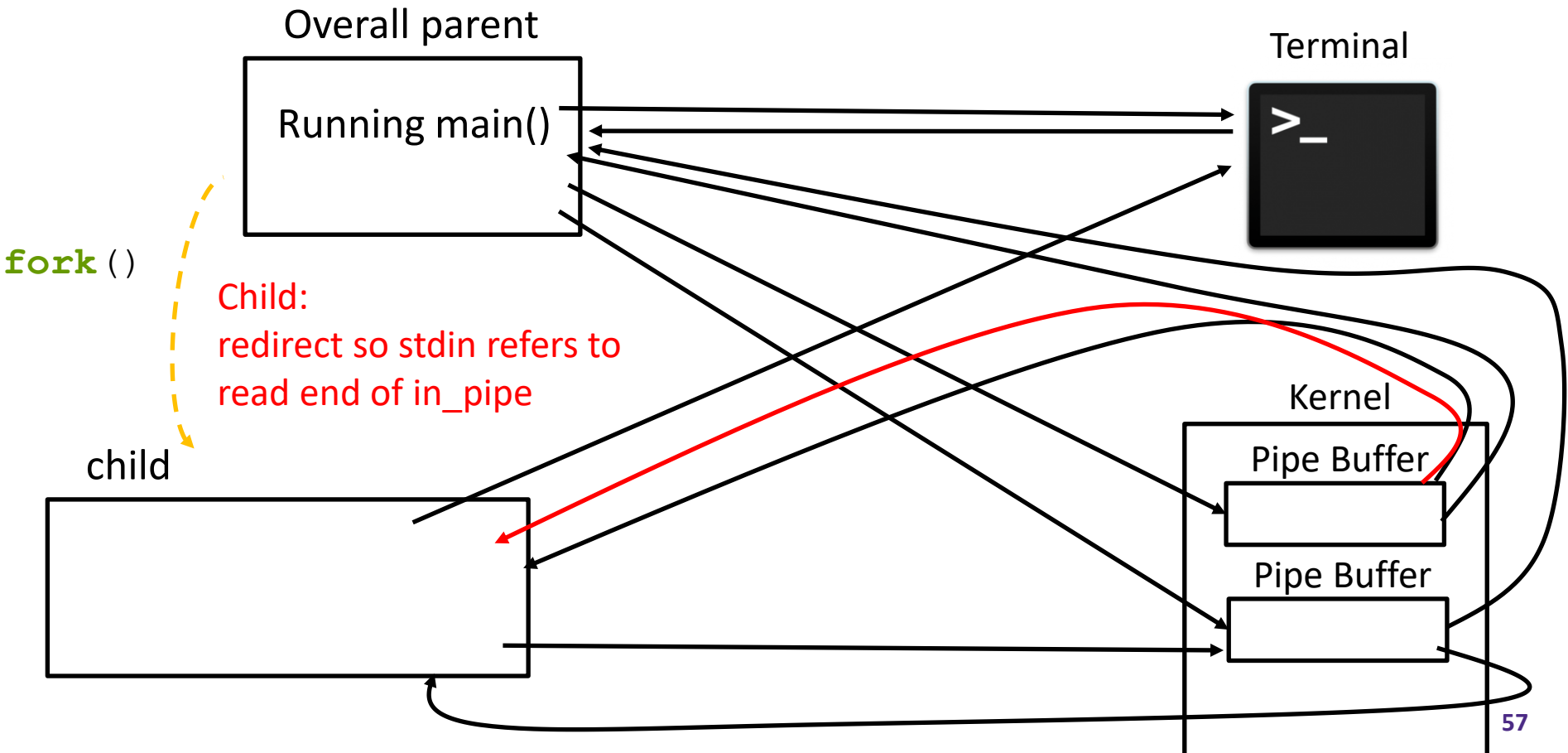
io_autograder.c Trace

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



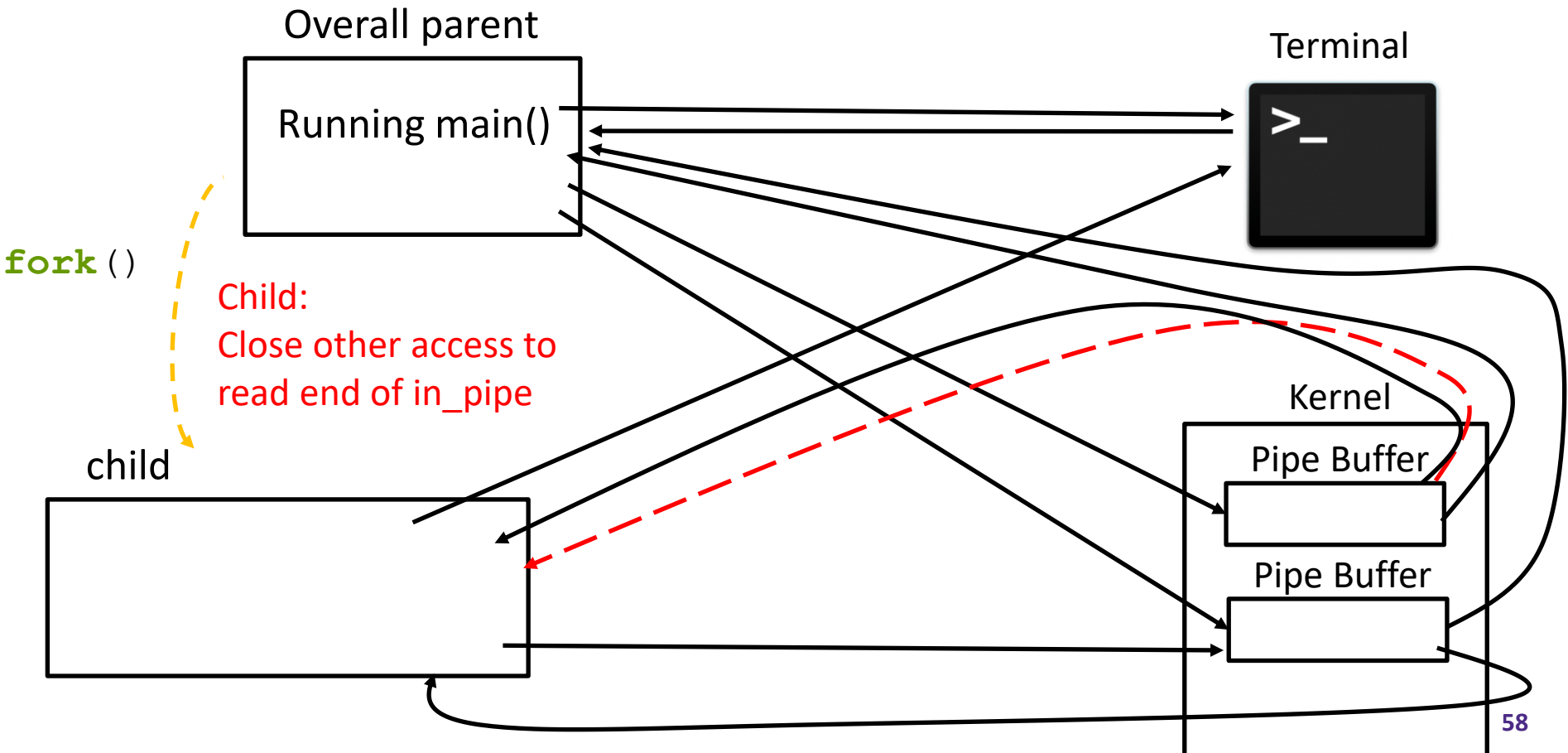
io_autograder.c Trace

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



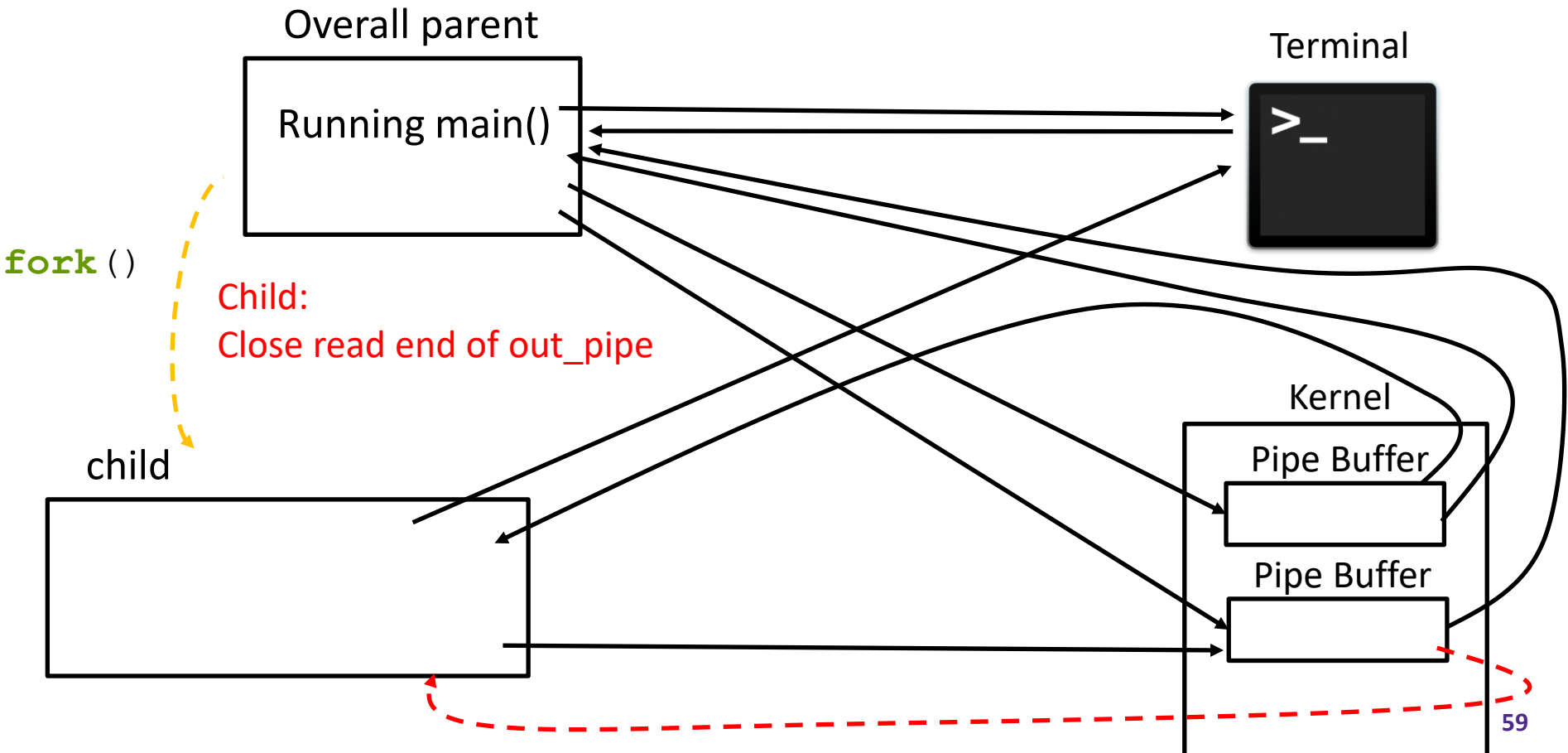
io_autograder.c Trace

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



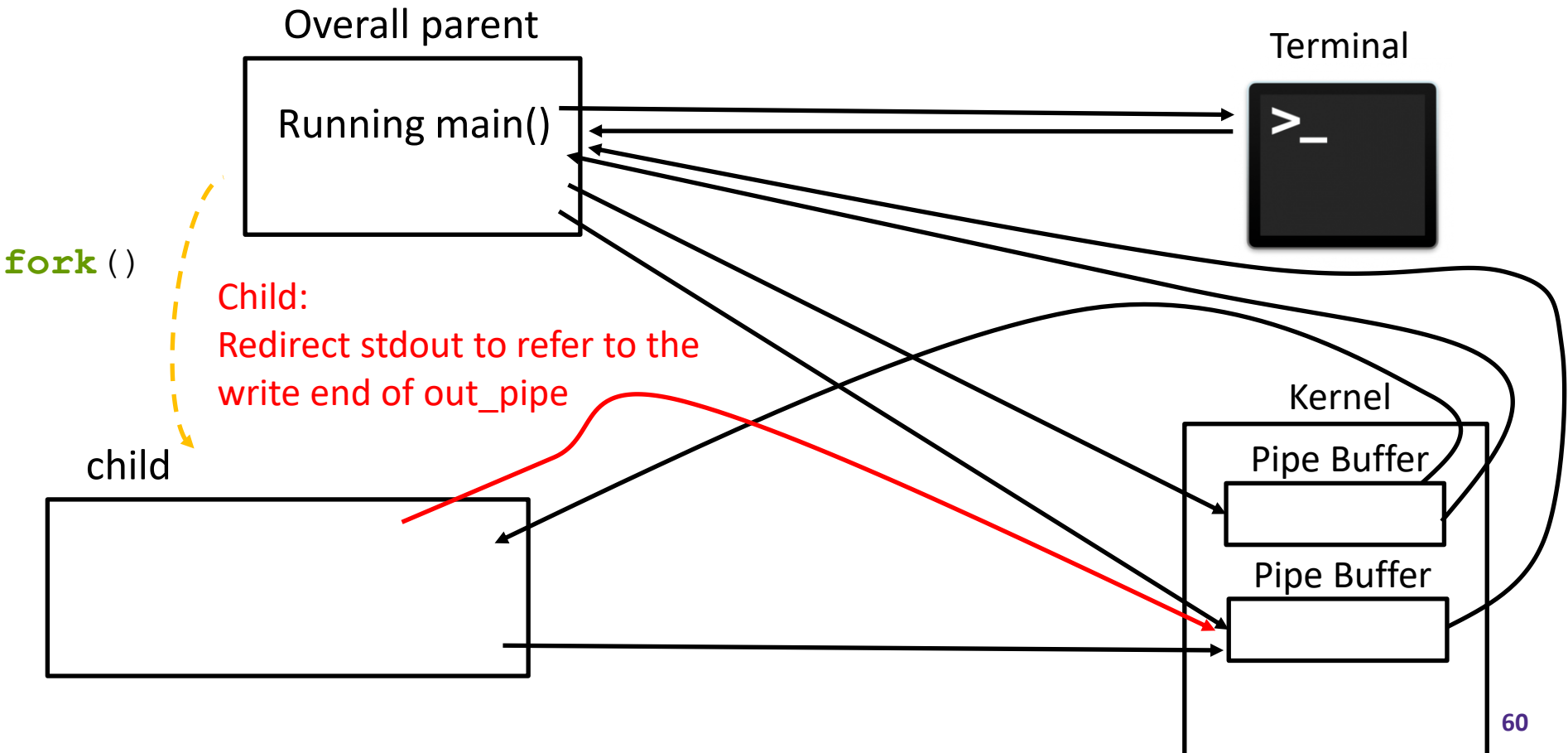
io_autograder.c Trace

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



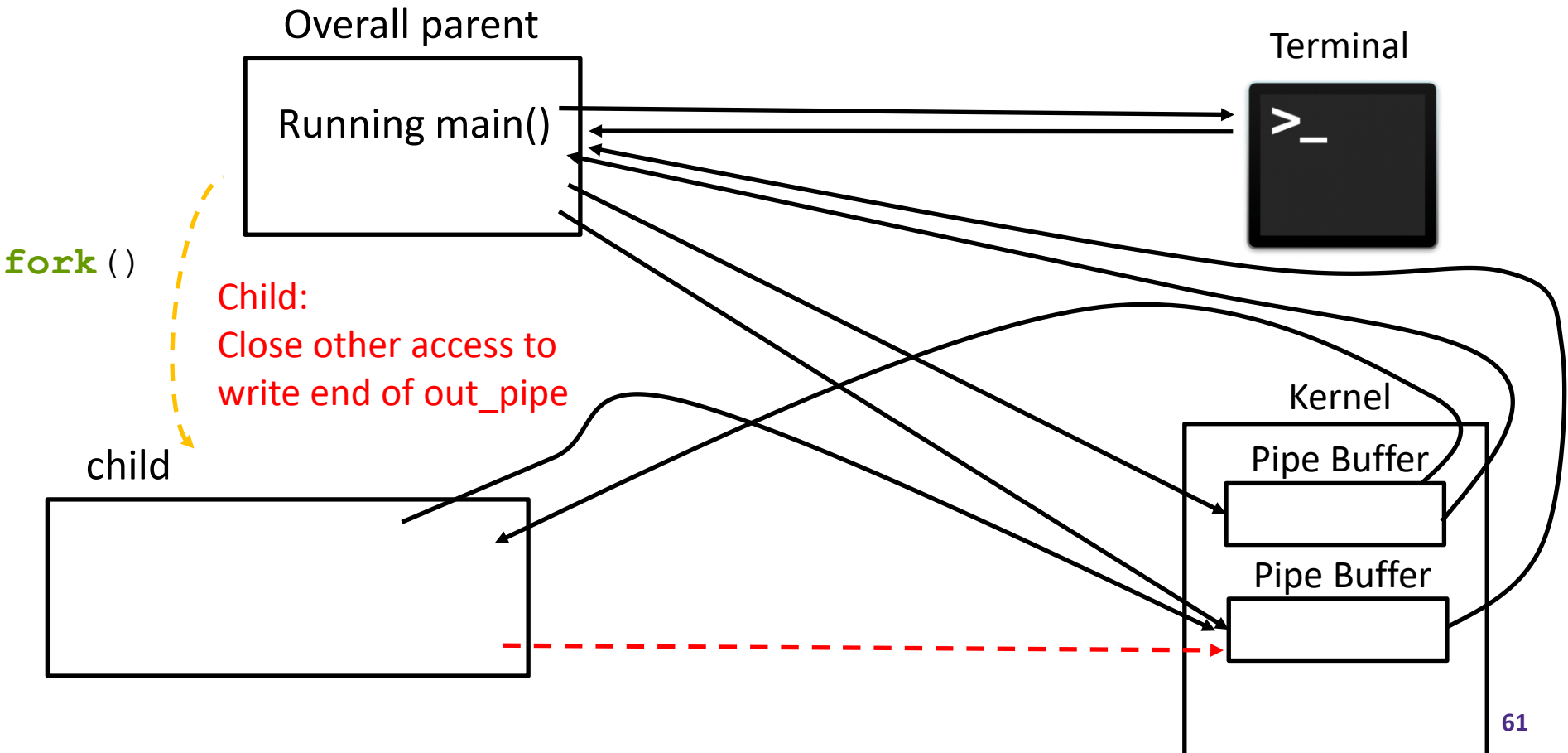
io_autograder.c Trace

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



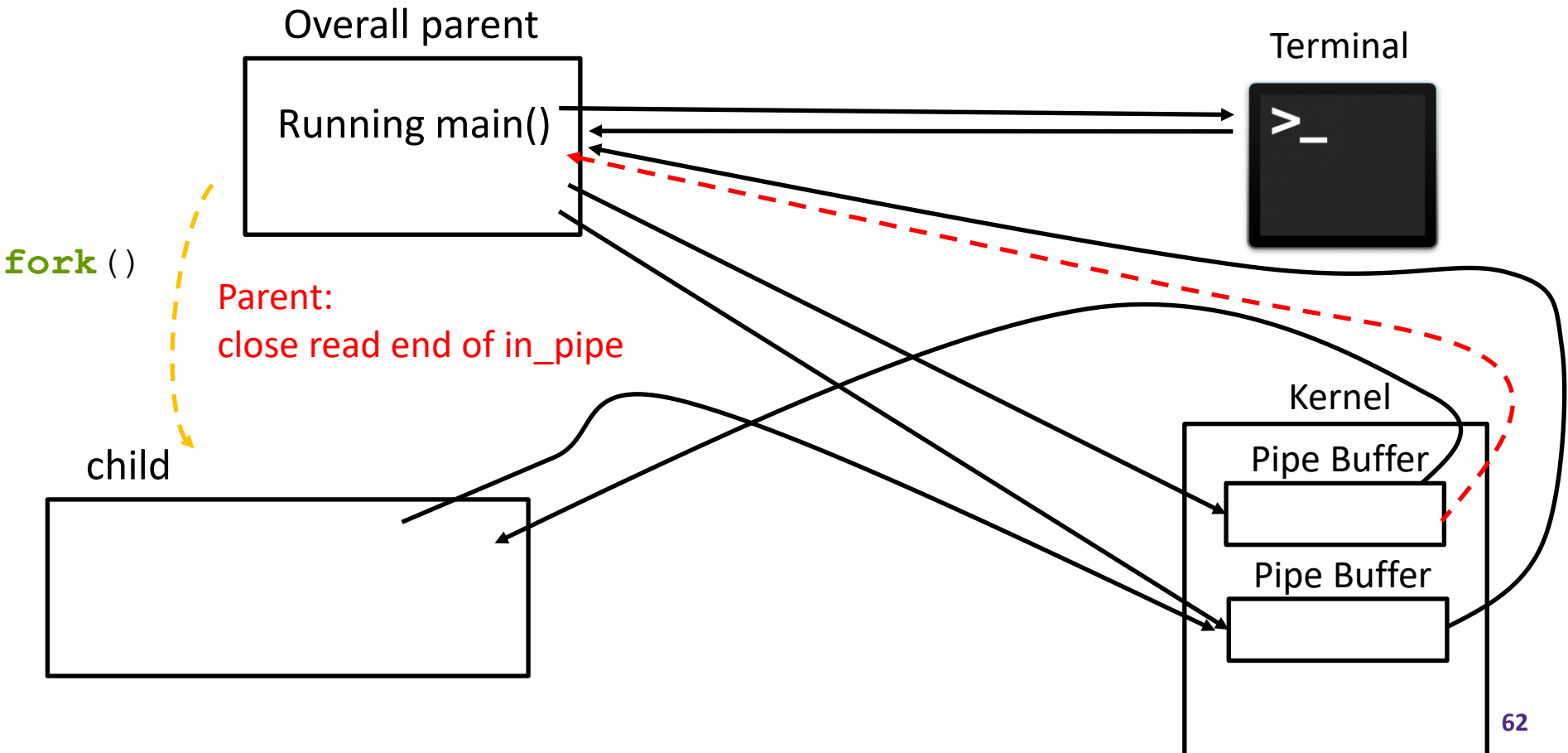
io_autograder.c Trace

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



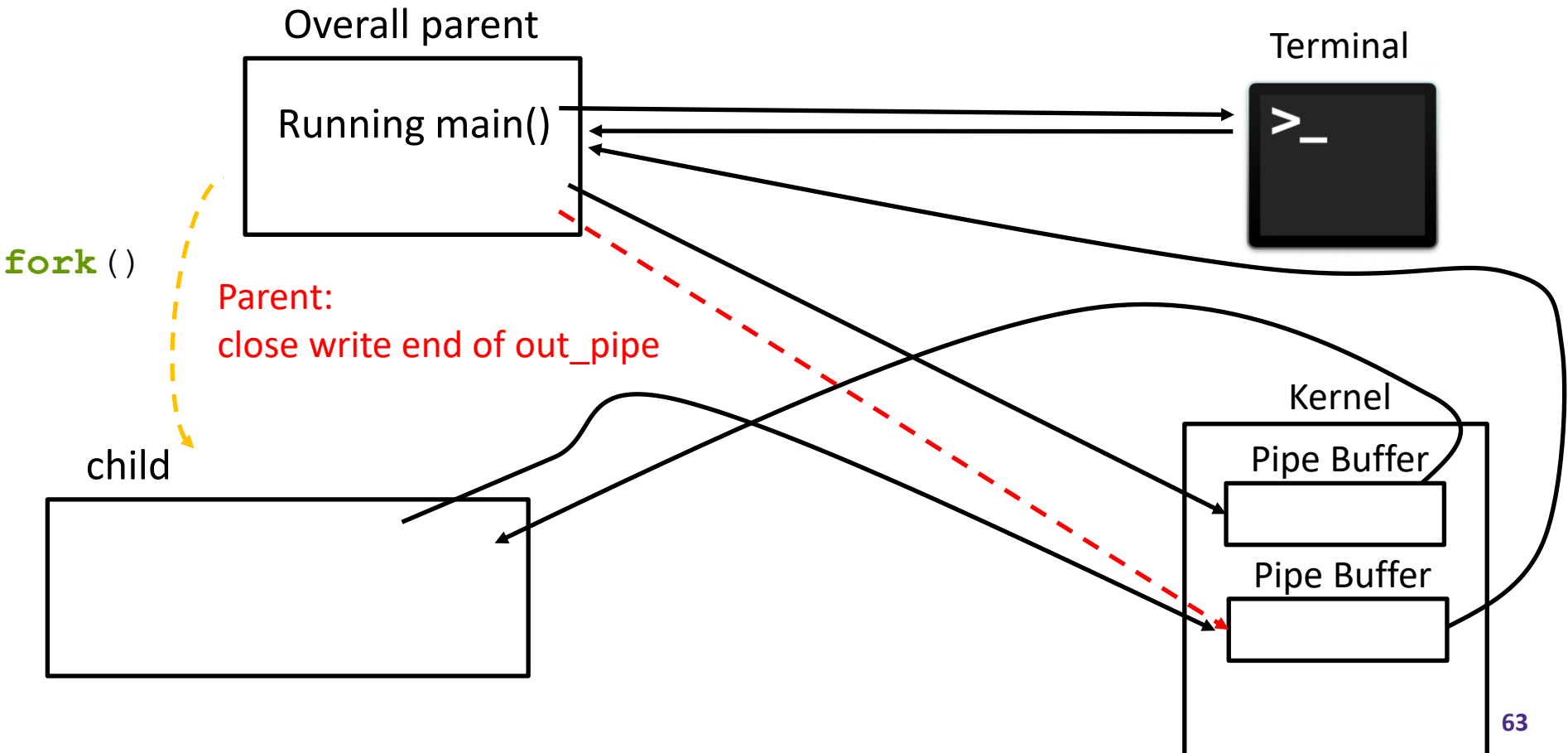
io_autograder.c Trace

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



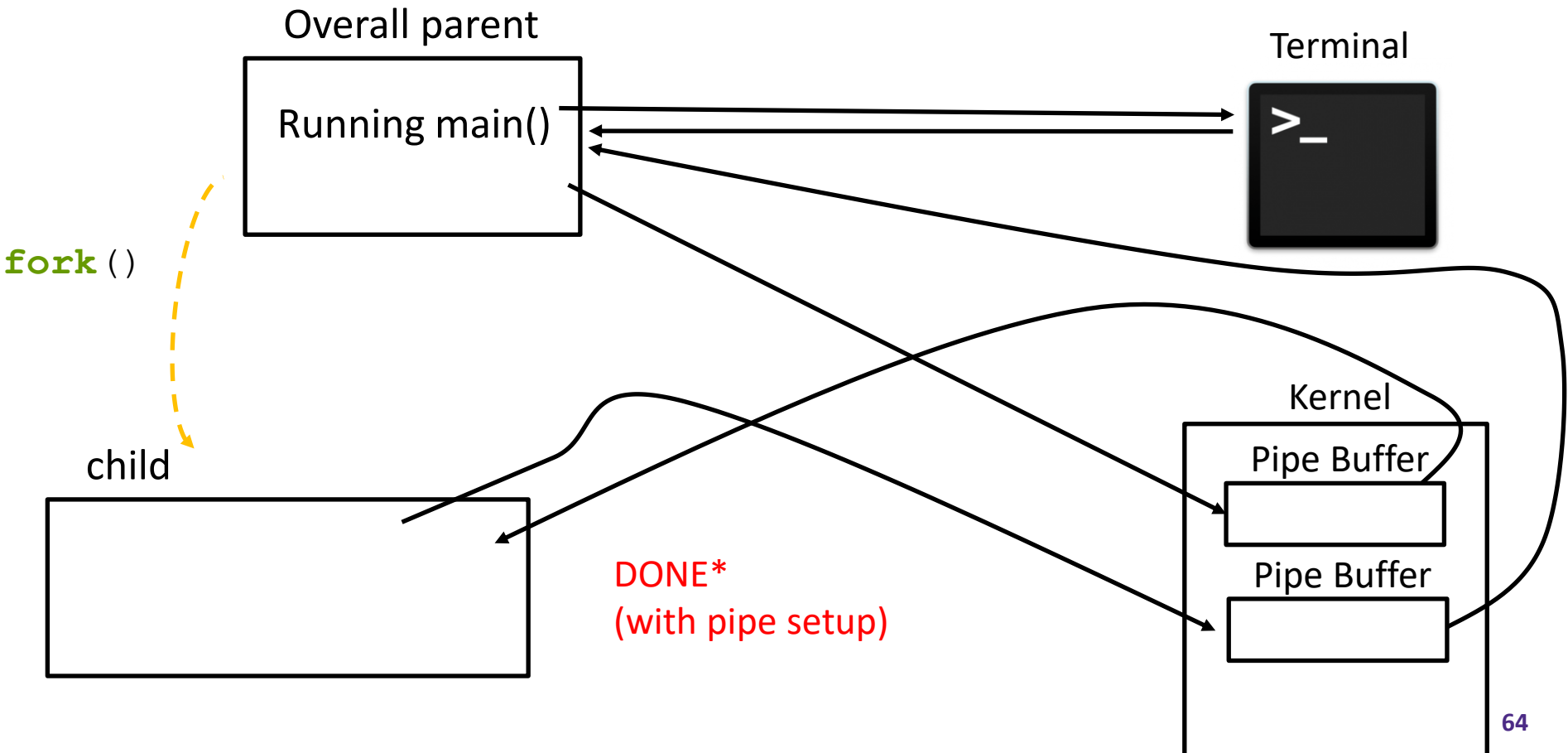
io_autograder.c Trace

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



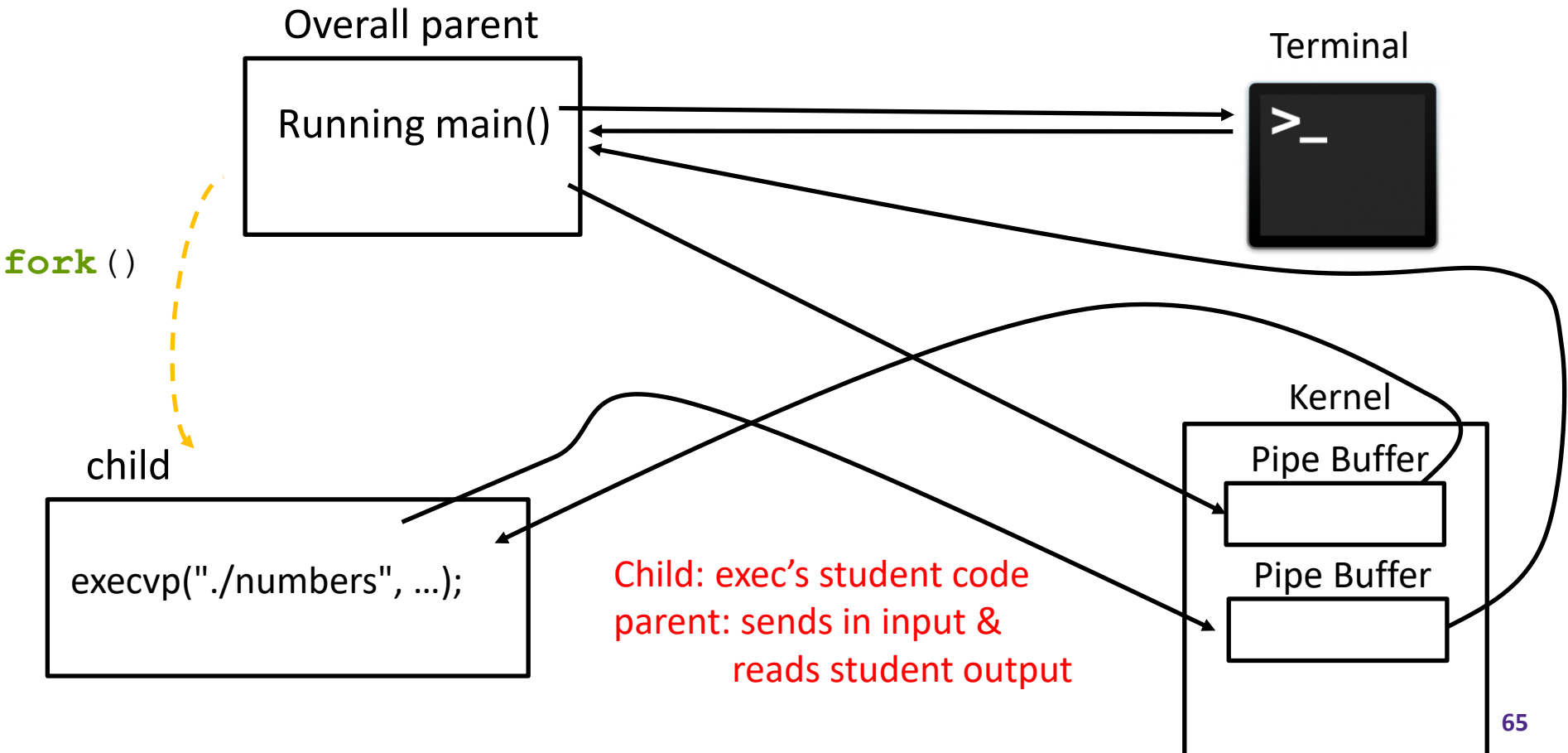
io_autograder.c Trace

- 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/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.

- <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
```

Poll Everywhere

pollev.com/tqm

- Which of the following commands will print the number of files in the current directory?

`cd`: change directory

`ls`: list directory contents

`wc`: reads from stdin, prints the number of words, lines, and characters read.

A. `ls > wc`
B. `cd . && ls wc`
C. `ls | wc`
D. `ls && wc`
E. **The correct answer is not listed**
F. **We're lost…**

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.