Files & File Descriptors
Computer Operating Systems, Fall 2023

Instructor: Travis McGaha

Head TAs: Nate Hoaglund   &   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	

Poll Everywhere

pollev.com/tqm

❖ Is there anything you would want to be taught in recitation, say if we had one next week?

Administrivia

❖ Proj0 (penn-shredder) Due 09/13 @ 11:59 pm

- ▪ This includes git & docker setup instructions. Do this part ASAP, it can take a while to debug issues with setup
- ▪ **This assignment is done on your own**

❖ Check-in Quiz 0 Due before this lecture

- ▪ Still open, to account for students joining the course a bit late
- ▪ Don't expect this to be true with future quizzes

❖ Check-in Quiz 1 Due in ~1 week

- ▪ Should be released tomorrow, will be on file descriptors

❖ I have OH this Friday 12pm – 2pm, Levine 269A

- ▪ (not recurring as of now)
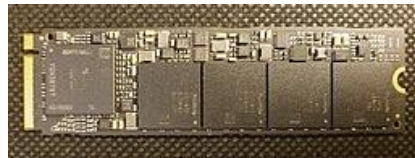
Poll Everywhere

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

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);
```
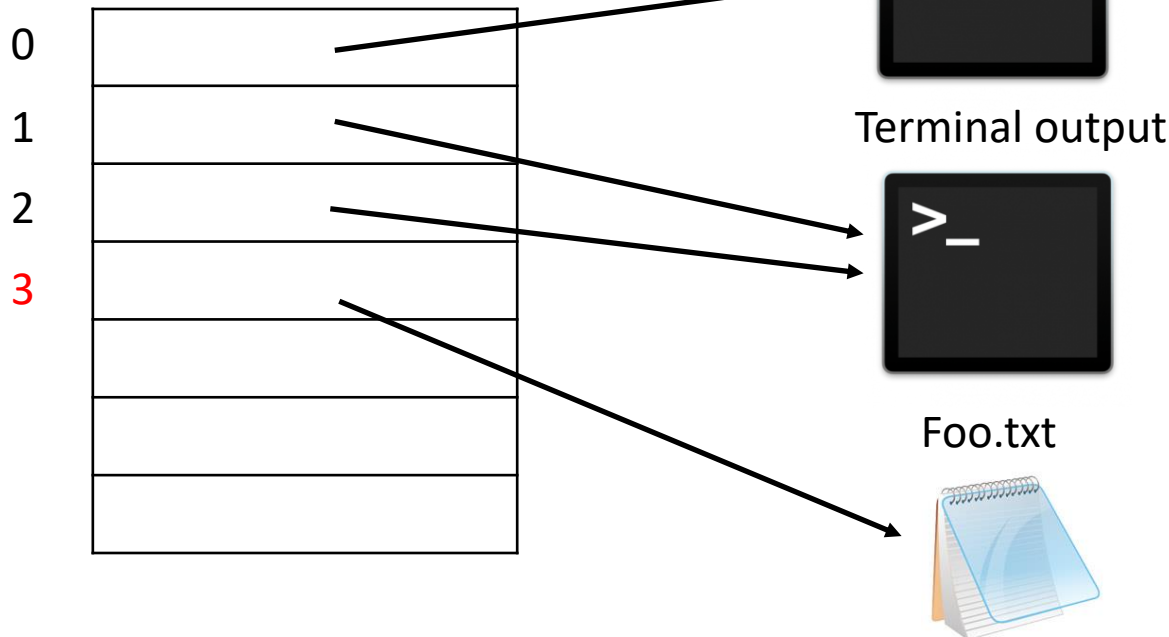
Lecture Outline

❖ 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 **it's own file descriptor table** managed by the OS

❖ The table is just an array, and the file descriptor is an index into it.
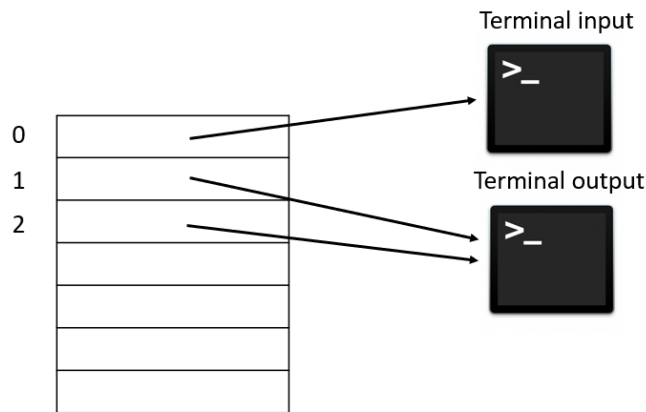
Terminal input

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

0

1

2

3

Terminal output

Foo.txt

Poll Everywhere

pollev.com/tqm

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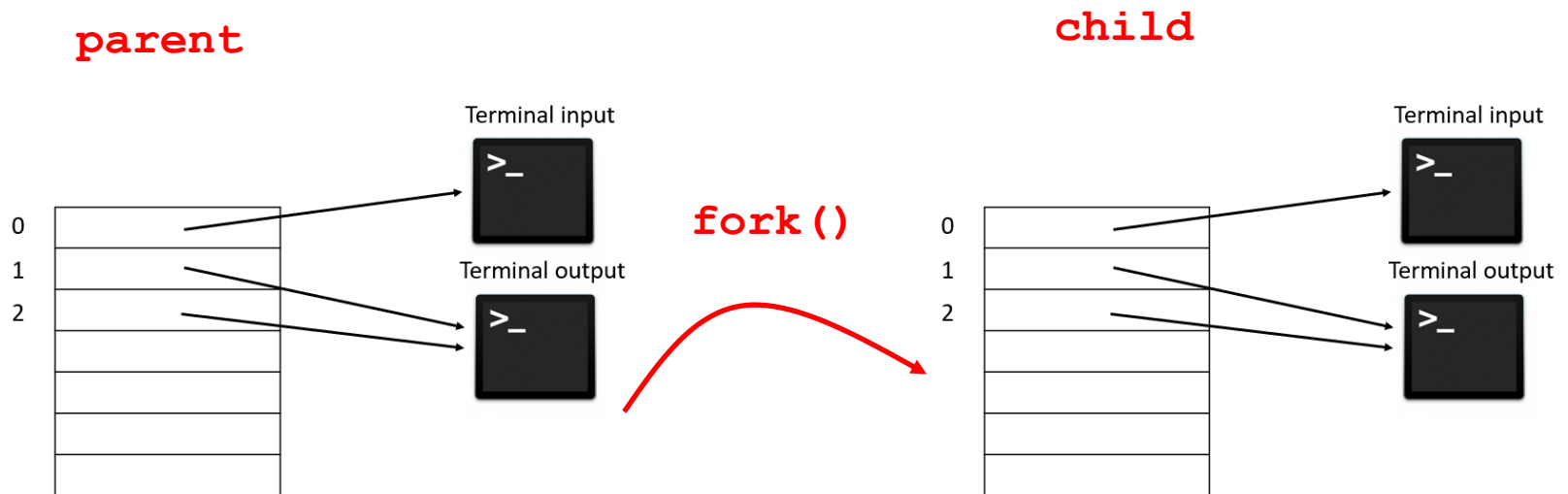 What negative affects may this have?

File Descriptor Table: Per Process

❖ each process will have **it's own file descriptor table** managed by the OS

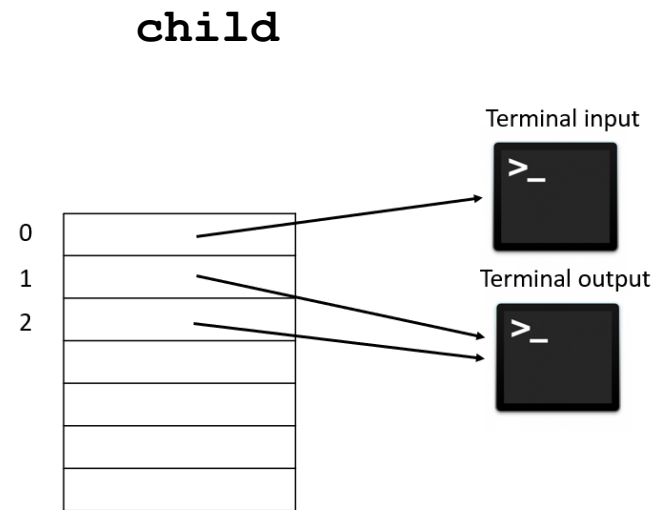❖ Fork will make a copy of the parent's file descriptor table for the child

File Descriptor Table: Per Process

❖ each process will have **it's own file descriptor table** managed by the OS

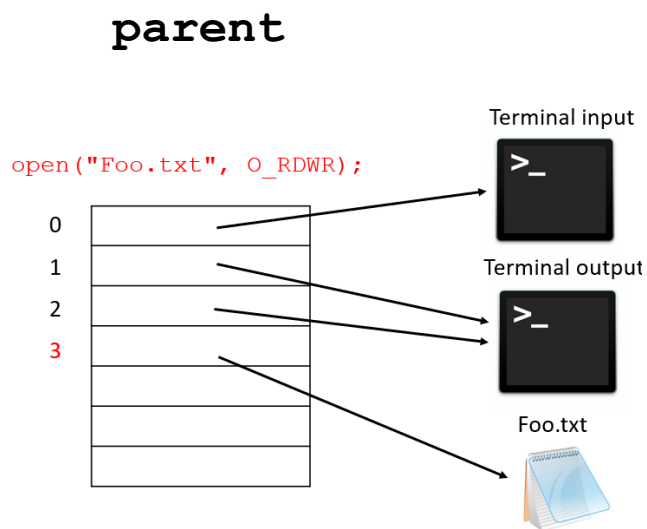❖ Fork will make a copy of the parent's file descriptor table for the child

File Descriptor Table: Per Process

❖ each process will have **it's own file descriptor table** managed by the OS

❖ Fork will make a copy of the parent's file descriptor table for the child
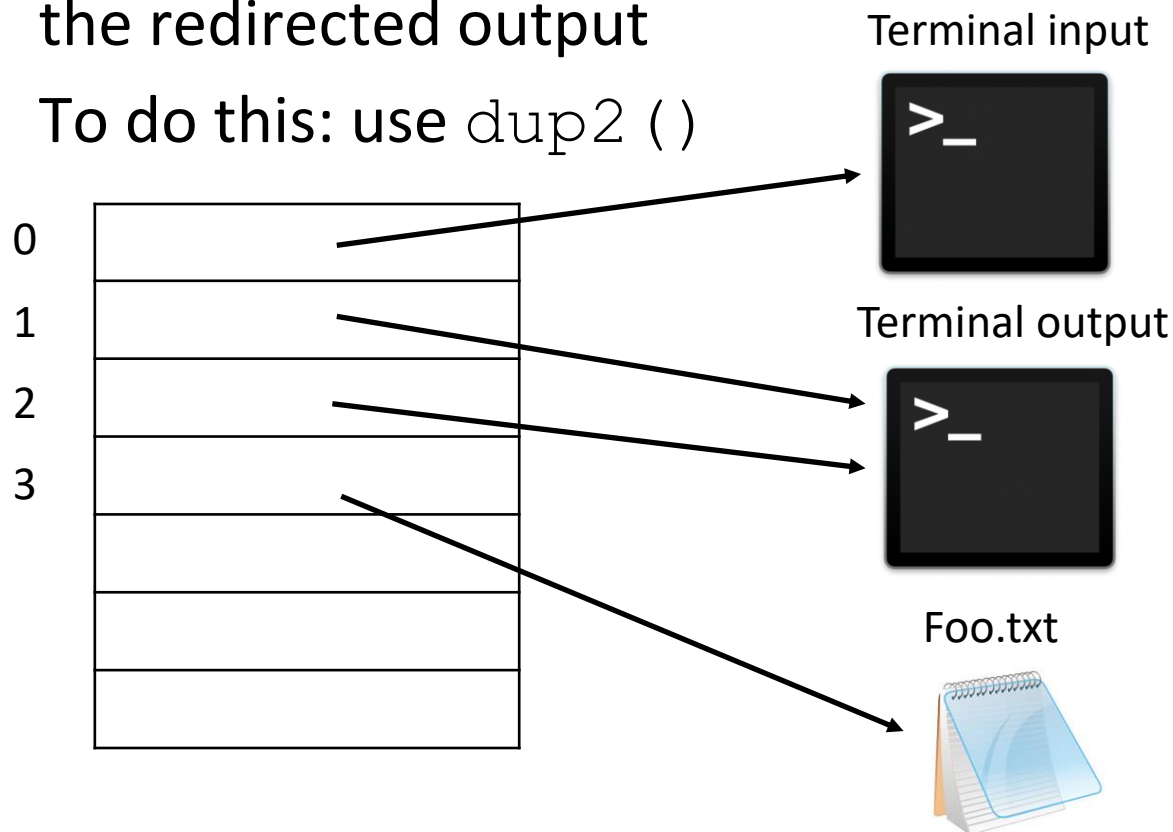

Child is unaffected by parent calling open!

Lecture Outline

❖ Intro to file descriptors

❖ File Descriptors: Big Picture

❖ **Redirection & Pipes**

❖ Unix Commands & Controls

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()`
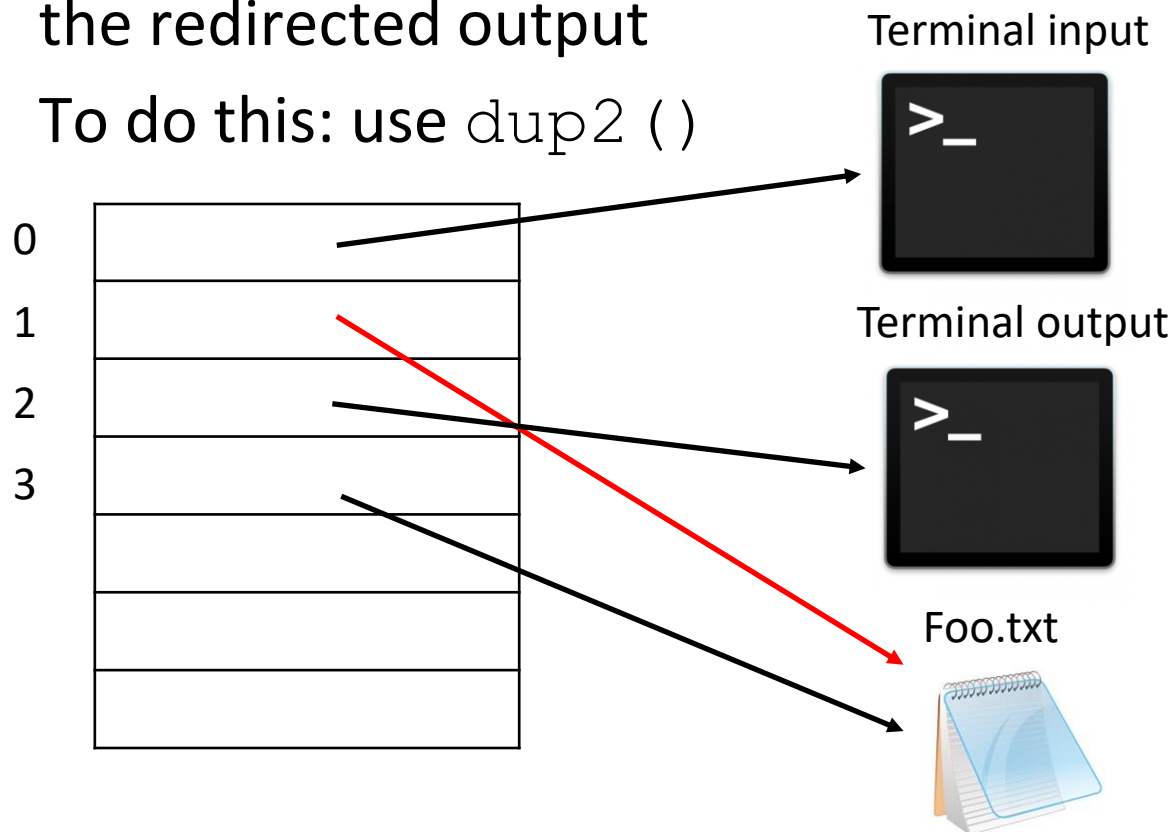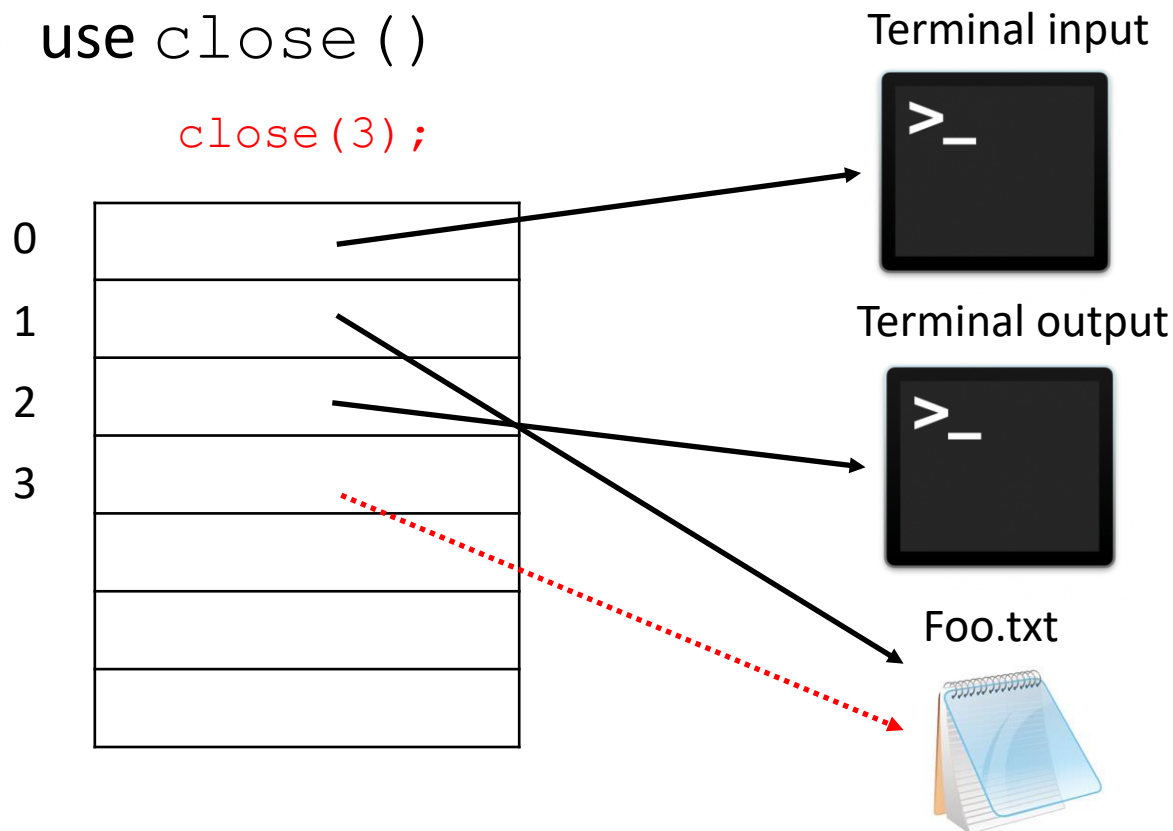
Terminal input

Terminal output

Foo.txt

0

1

2

3

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()`

Terminal input

Terminal output

Foo.txt

0
1
2
3

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);`

Terminal input
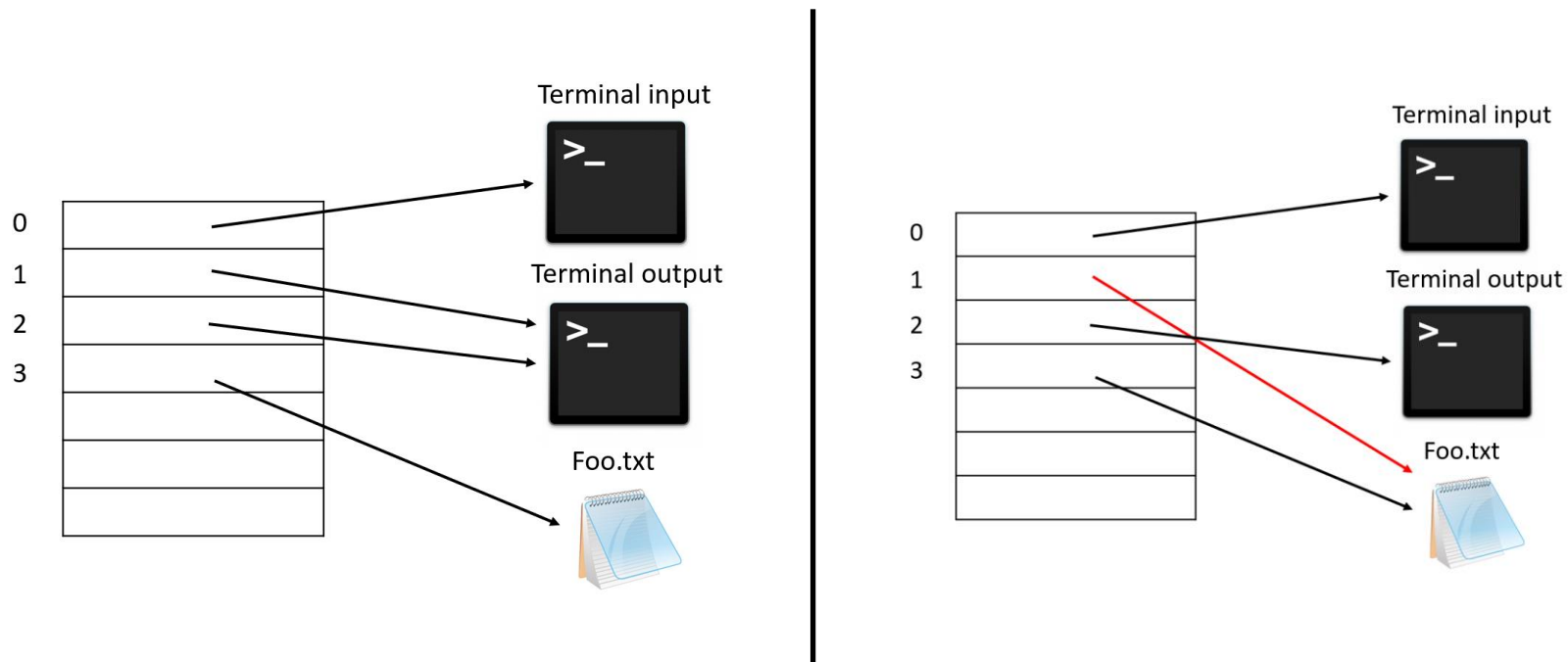
Terminal output

Foo.txt

0

1

2

3

dup2

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

❖ Look it up in the man pages ☺

📊 Poll Everywhere

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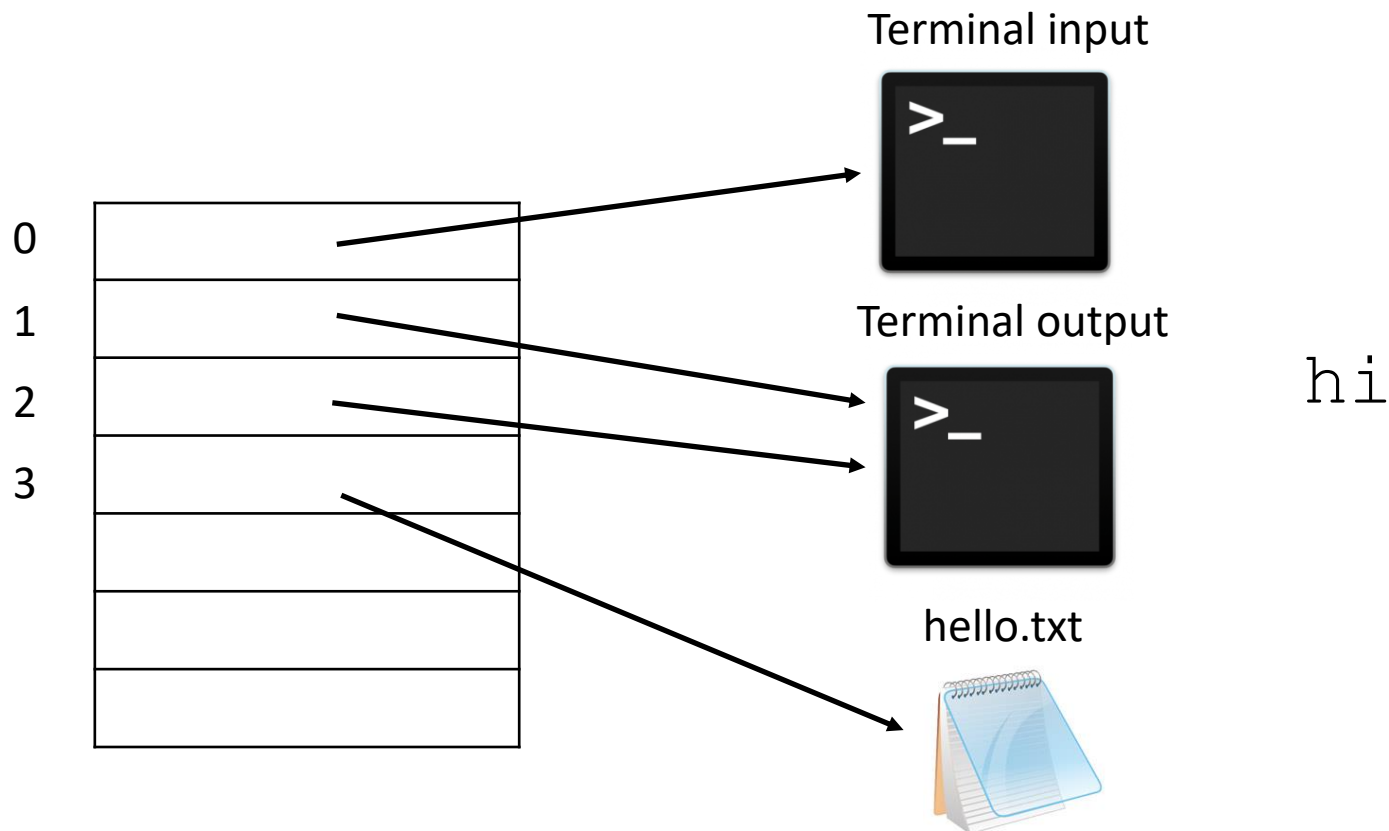
Poll Everywhere

pollev.com/tqm

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

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

Explanation

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

Terminal input

Terminal output

hi

hello.txt

0

1

2

3

Explanation

```
close(STDOUT_FILENO);
```


Terminal input

Terminal output

hi

hello.txt

Explanation

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

Terminal input

Terminal output

hi

hello.txt

Explanation

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

Terminal input

Terminal output

hi

hello.txt

0

1

2

3

Explanation

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

Terminal input

Terminal output

`hi`

hello.txt

`!`

Explanation

`close``(fd);`

Terminal input

Terminal output

hi

hello.txt

!

Explanation

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

Terminal input

Terminal output

hi

hello.txt

!

*

0

1

2

3

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

31

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

Terminal input

Terminal output

Kernel

Pipe Buffer

◧ Poll Everywhere

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

```
 1 #include <unistd.h>
 2 #include <stdlib.h>
 3 #include <stdio.h>
 4
 5 int main() {
 6   int pipe_fds[2];
 7   pipe(pipe_fds);
 8
 9   pid_t pid = fork();
10
11   if (pid == 0) {
12     /// close my end of the pipe
13     close(pipe_fds[0]);
14
15     write(pipe_fds[1], "Hello!", 6);
16
17     char str[7];
18     ssize_t chars_read = read(pipe_fds[1], str, 6);
19
20     if (chars_read != -1) {
21       str[chars_read] = '\0';
22
23       printf("%s\n", str);
24     }
25
26     exit(EXIT_SUCCESS);
27   }
28   // parent
```

```
28   // parent
29
30   /// close my end of the pipe
31   close(pipe_fds[1]);
32
33   char str[7];
34   ssize_t chars_read = read(pipe_fds[0], str, 6);
35
36   if (chars_read != -1) {
37     str[chars_read] = '\0';
38     printf("%s\n", str);
39   }
40
41   write(pipe_fds[0], "Howdy!", 6);
42
43   return EXIT_SUCCESS;
44 }
45
```

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

Overall parent

```
┌─────────────────────┐
│                     │
│  Running main()     │  ⟵⟶  Terminal
│                     │
│                     │
└─────────────────────┘
```

Terminal

`io_autograder.c` Trace

❖ First:
   we compile the program with the gcc command

Overall parent

Running main()

fork()

Terminal

child

execvp("gcc", …);

`io_autograder.c` Trace

❖ First:

we compile the program with the gcc command

Overall parent

Terminal

Running main()

waitpid()

fork()

child

execvp("gcc", …);

`io_autograder.c` Trace

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

Overall parent

Terminal

Running main()

`io_autograder.c` Trace

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

Overall parent

Running main()

Terminal

Kernel

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

Pipe Buffer

`io_autograder.c` Trace

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

Overall parent

Running main()

Terminal

Kernel

Pipe Buffer

Pipe Buffer

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

One pipe to get the output
from user program

`io_autograder.c` Trace

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

Overall parent

Running main()

Terminal

fork()

Fork to create the process
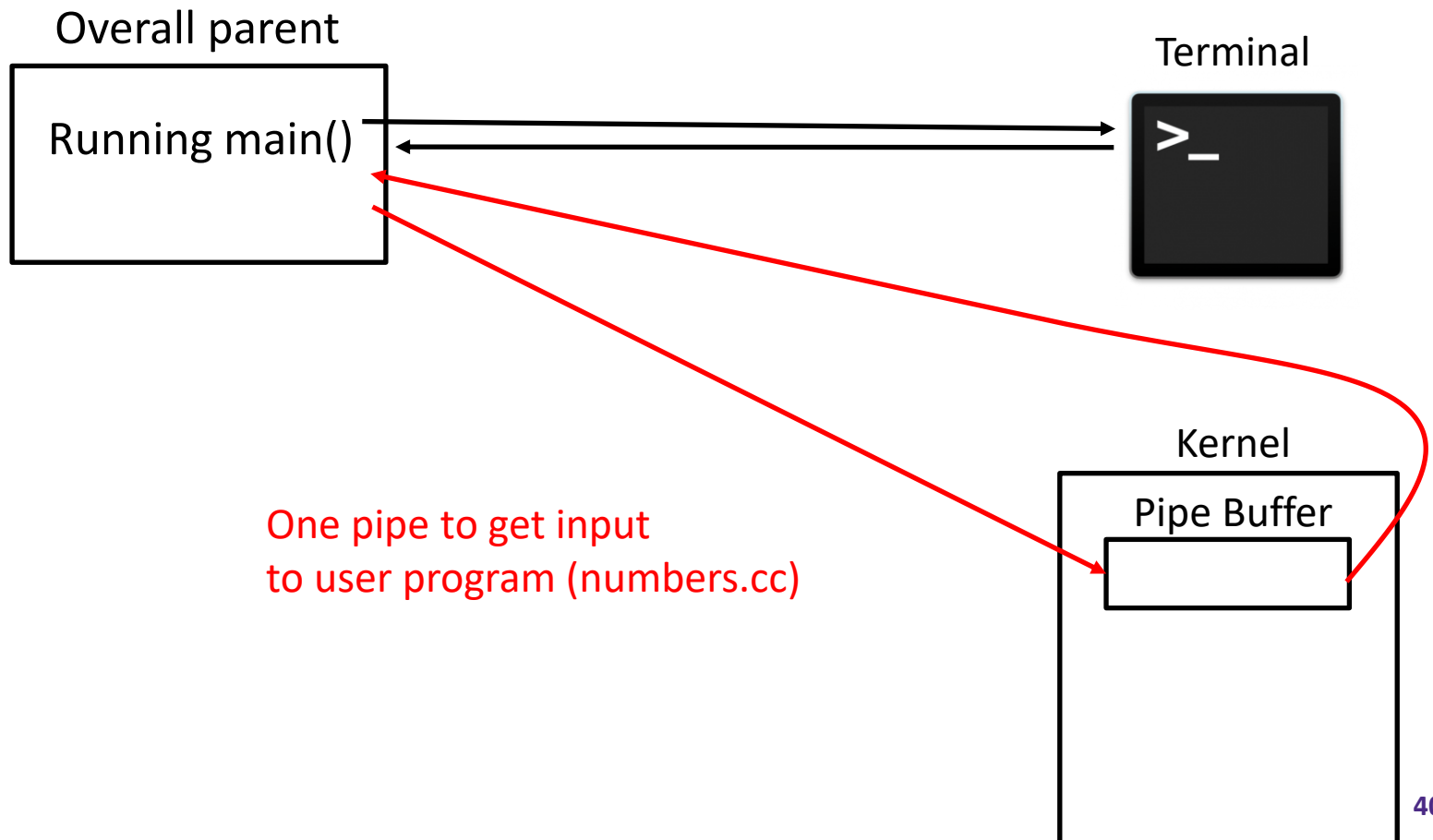that willrun the student's code

child

Kernel

Pipe Buffer

Pipe Buffer

`io_autograder.c` Trace

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

Overall parent

Running main()

Terminal

fork()

Child:
close write end of in_pipe

child

Kernel

Pipe Buffer

Pipe Buffer

`io_autograder.c` Trace

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

Overall parent

Running main()

Terminal

fork()

Child:
redirect so stdin refers to
read end of in_pipe
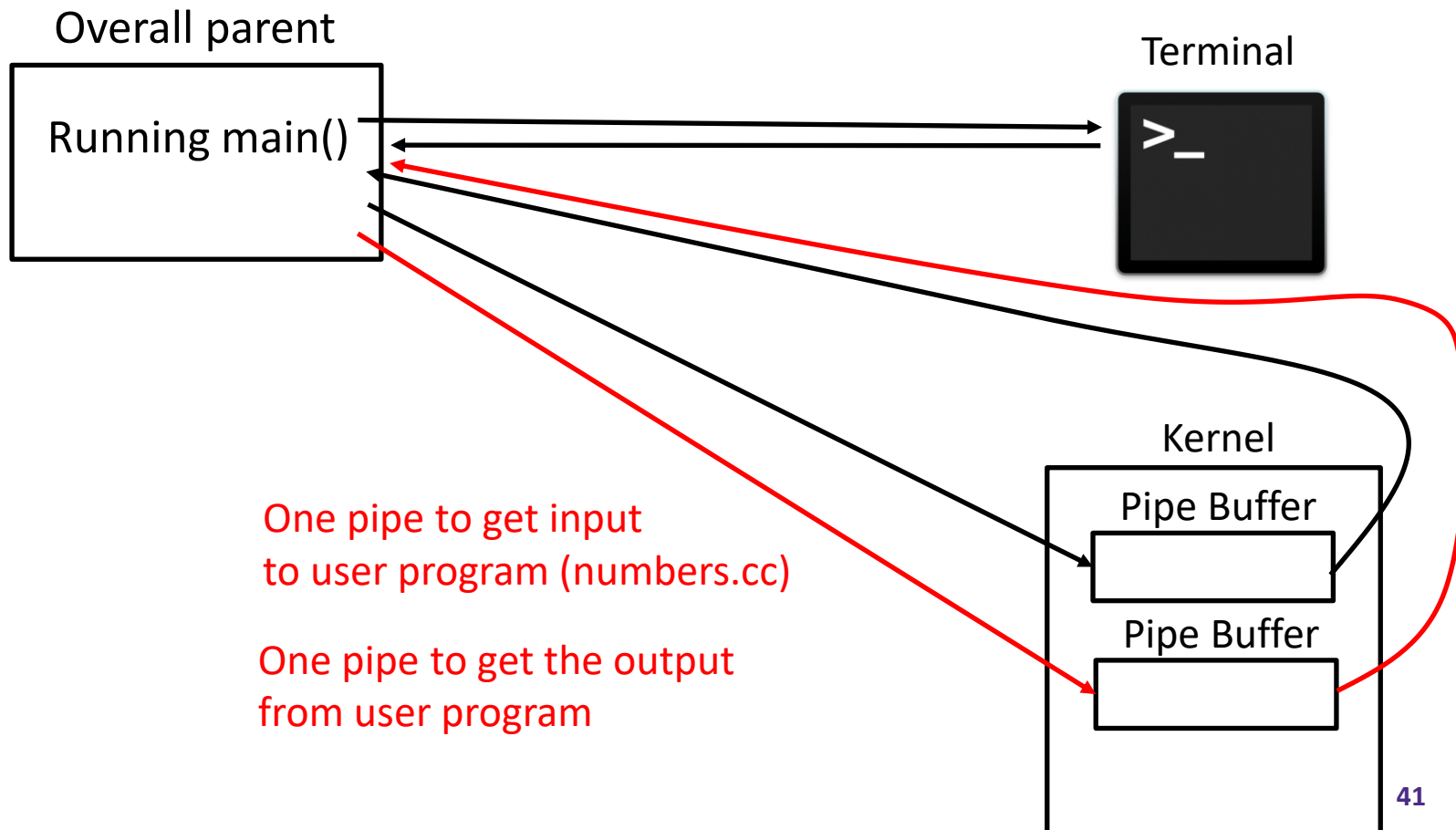
child

Kernel

Pipe Buffer

Pipe Buffer

`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…
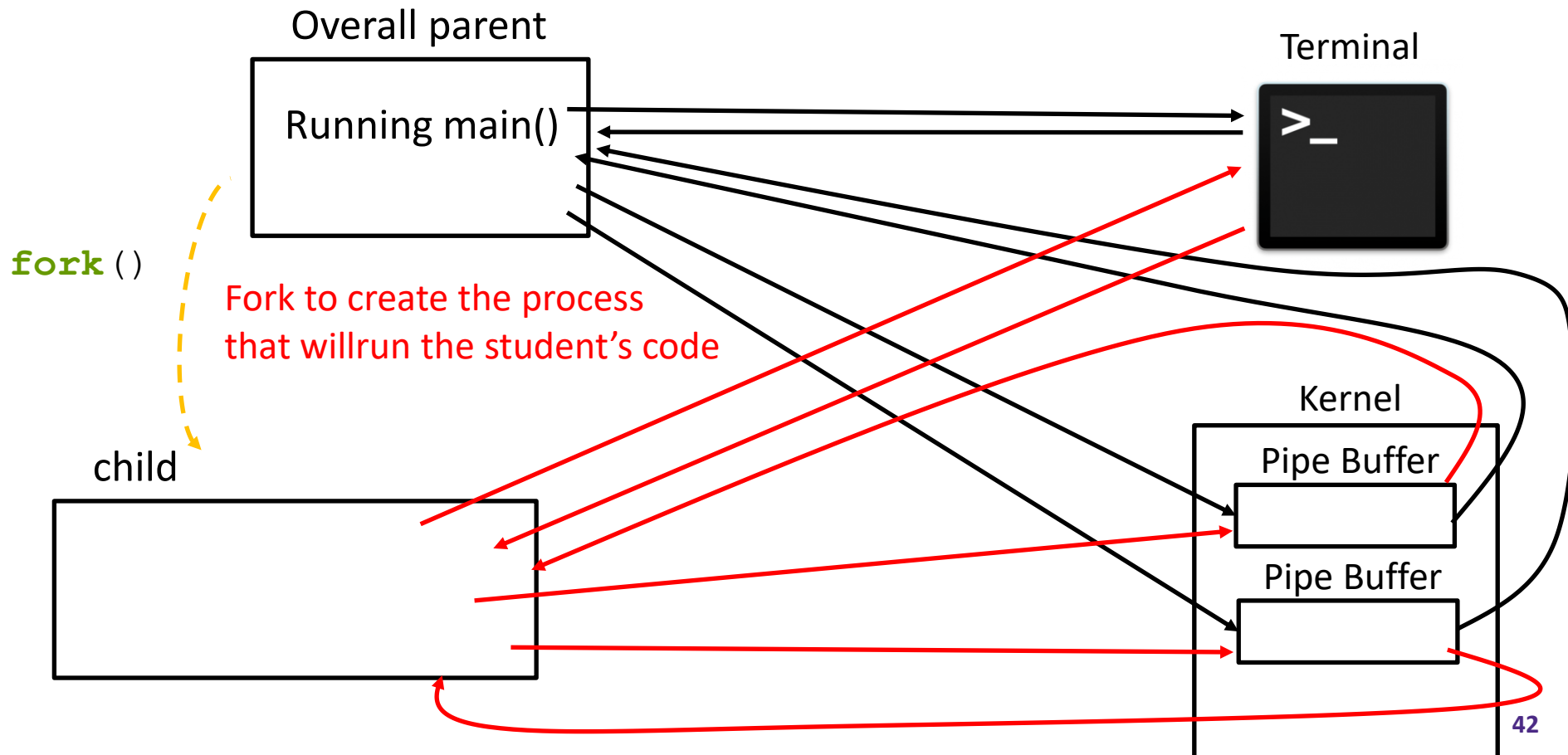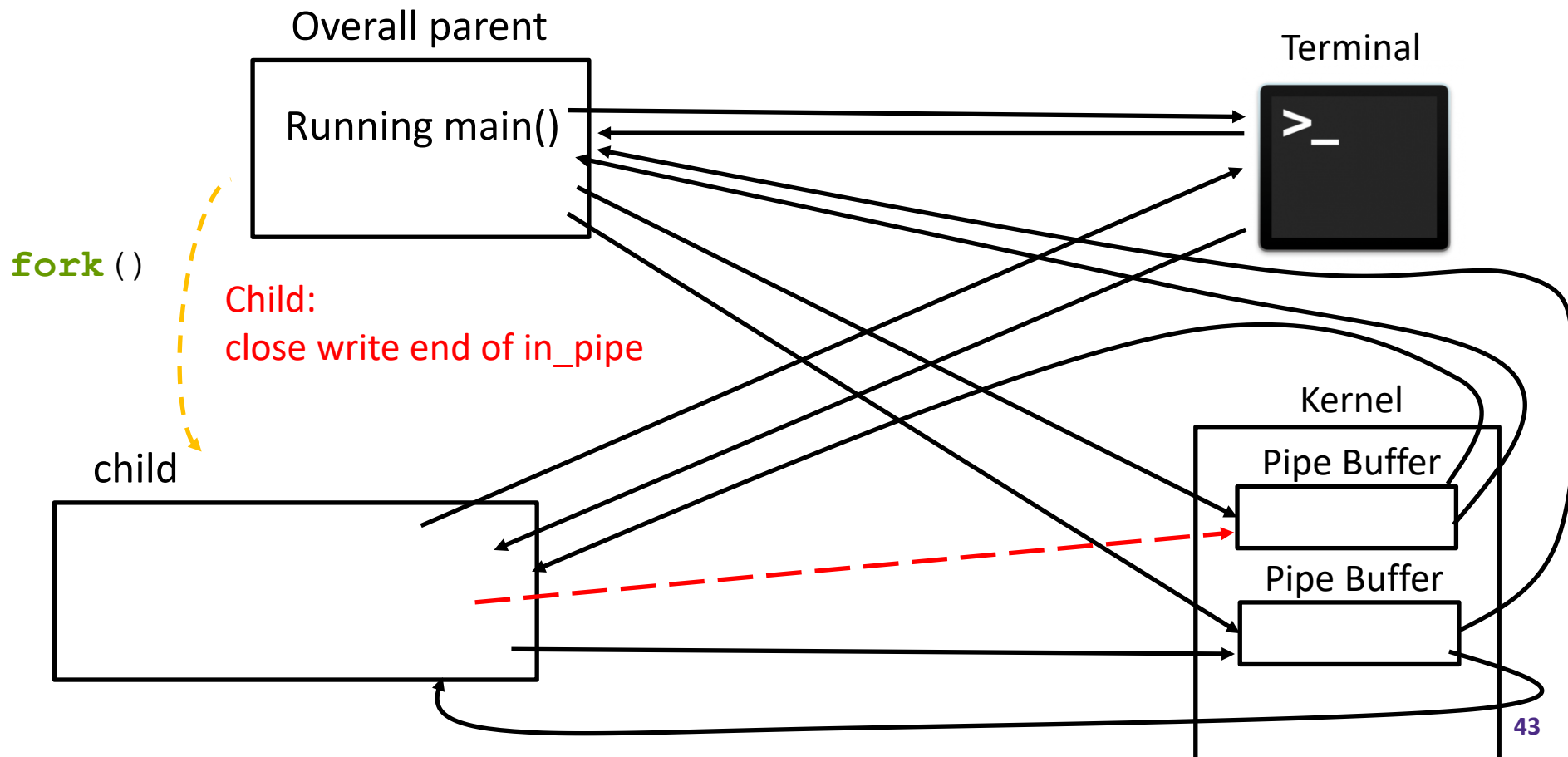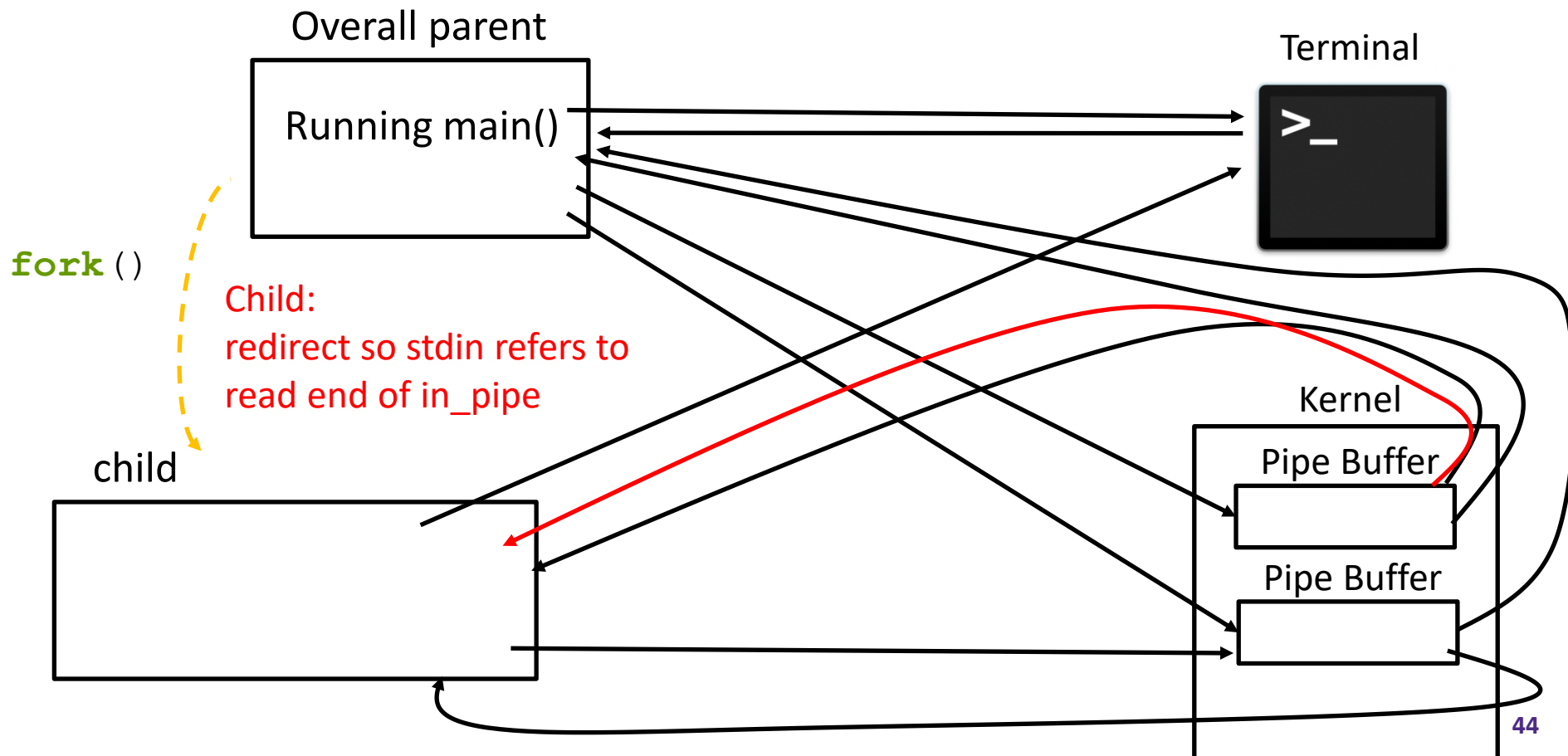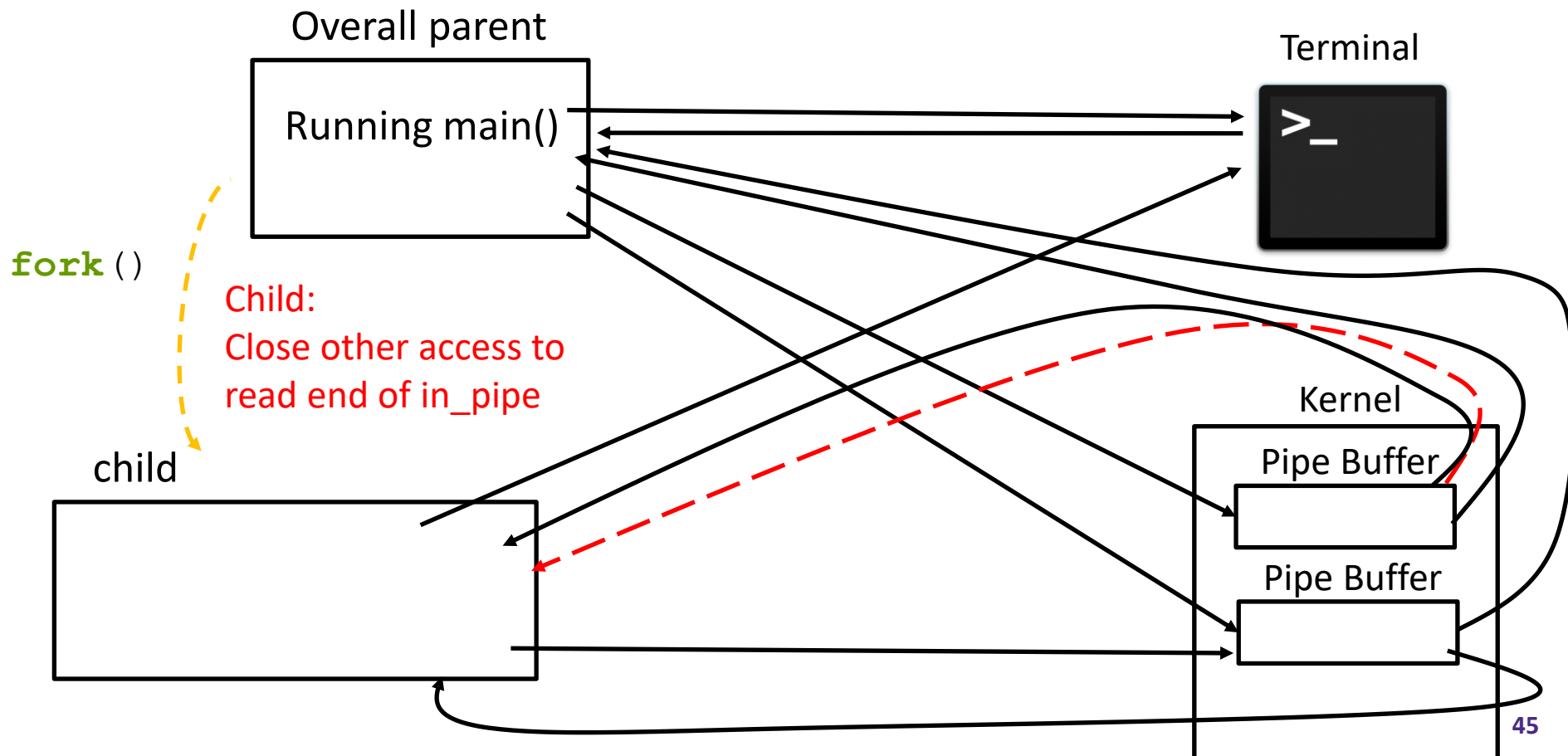 BUT send autograder input and capture output

Overall parent

Running main()

fork()

Child:
Close read end of out_pipe

child

Terminal

Kernel

Pipe Buffer

Pipe Buffer

46

`io_autograder.c` Trace

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

Overall parent

Running main()

Terminal

fork()

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

child

Kernel

Pipe Buffer

Pipe Buffer

`io_autograder.c` Trace

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

Overall parent

Running main()

Terminal

fork()

Child:
Close other access to
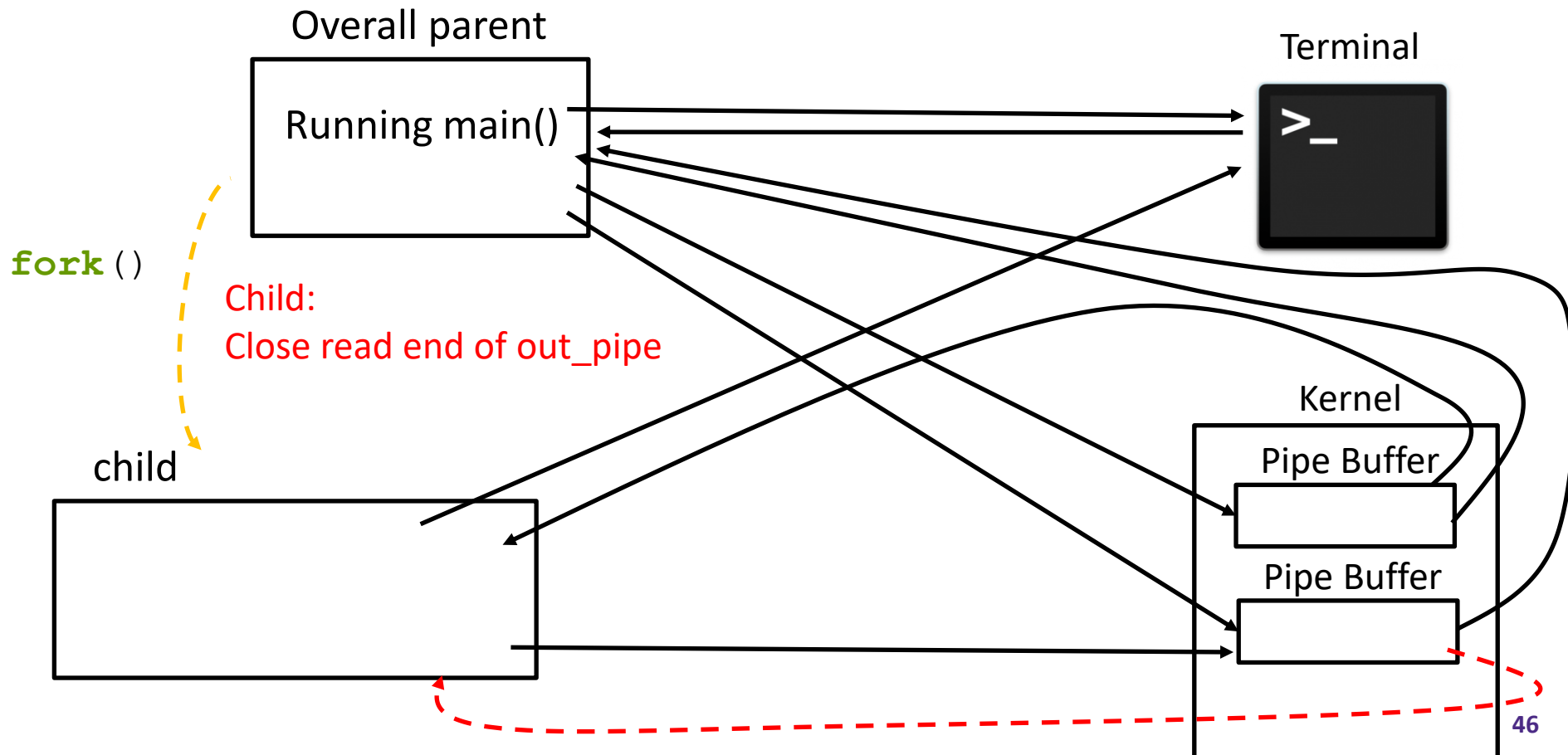write end of out_pipe

child

Kernel

Pipe Buffer

Pipe Buffer

48

`io_autograder.c` Trace

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

Overall parent

Running main()

Terminal

fork()

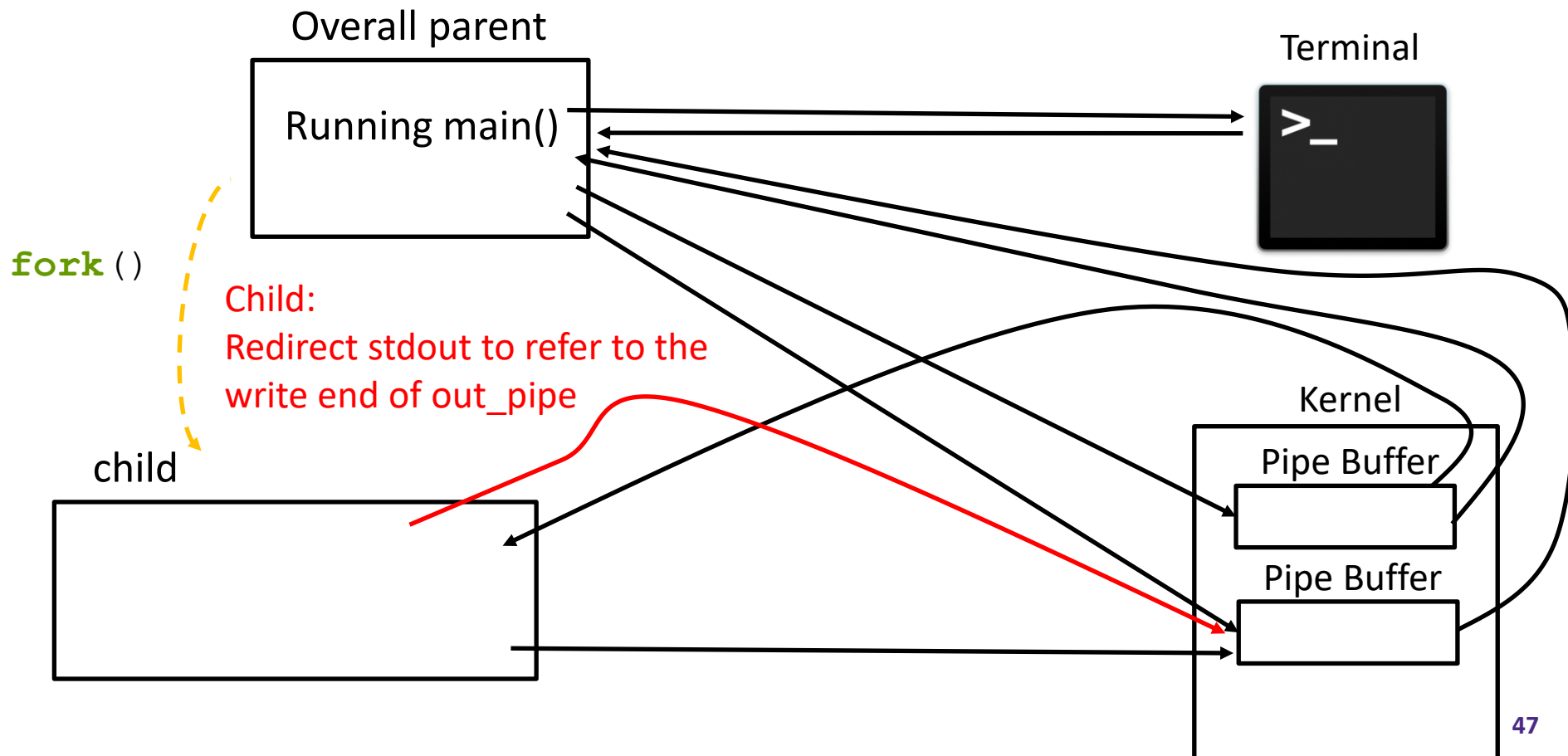Parent:
close read end of in_pipe

child

Kernel

Pipe Buffer

Pipe Buffer

`io_autograder.c` Trace

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

Overall parent

Running main()

Terminal

fork()

Parent:
close write end of out_pipe

child

Kernel

Pipe Buffer

Pipe Buffer

`io_autograder.c` Trace

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

Overall parent

Running main()

Terminal

fork()

child

DONE*
(with pipe setup)

Kernel

Pipe Buffer

Pipe Buffer

`io_autograder.c` Trace

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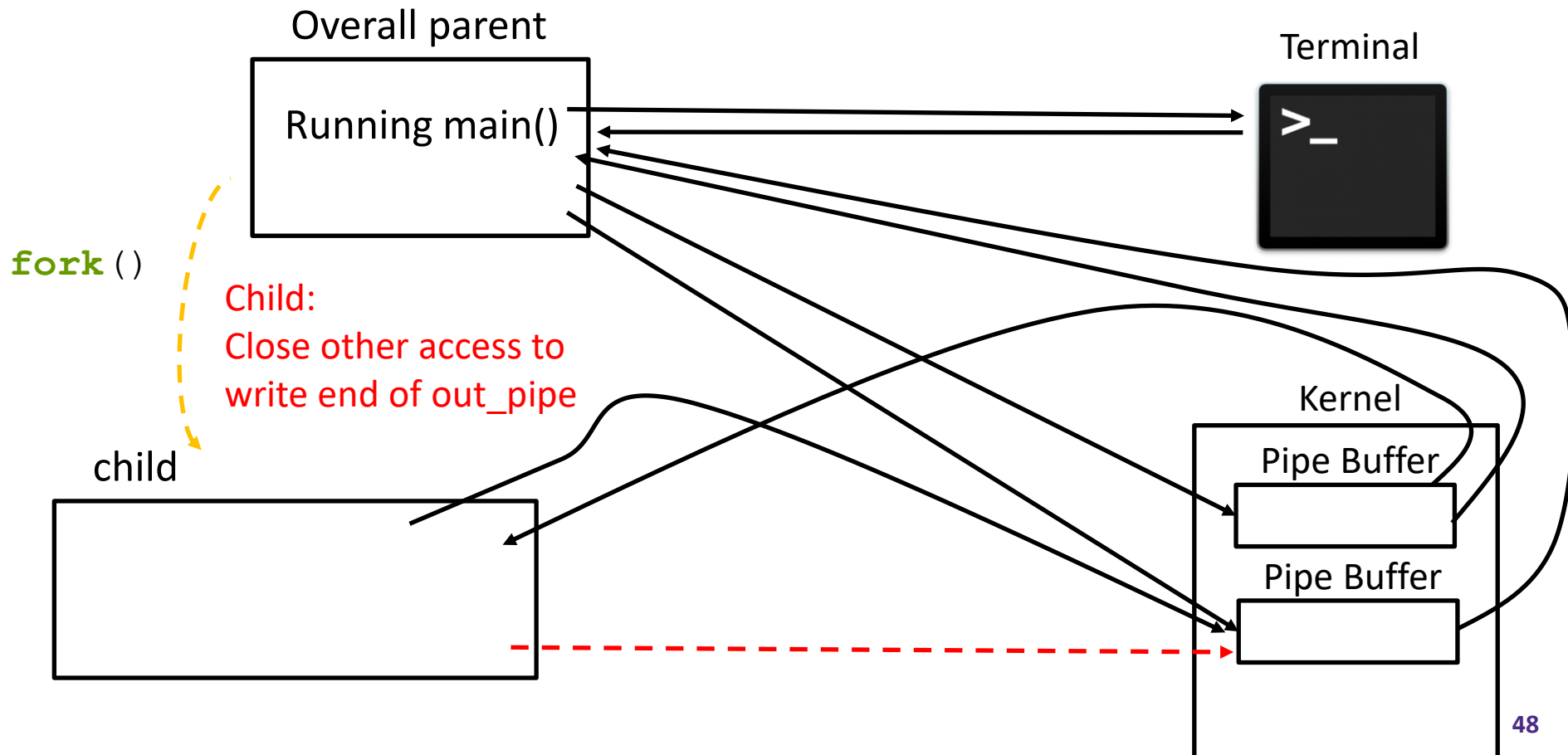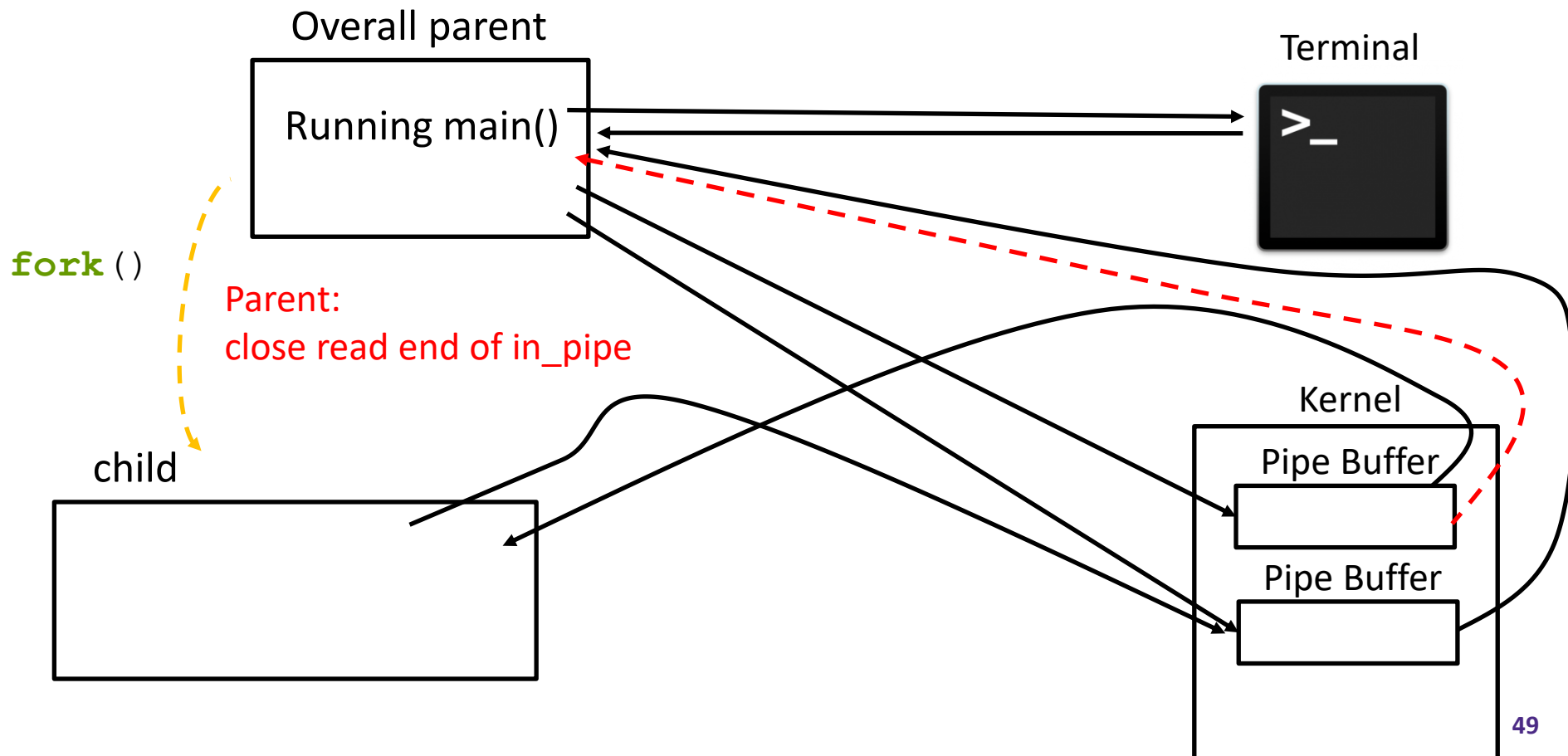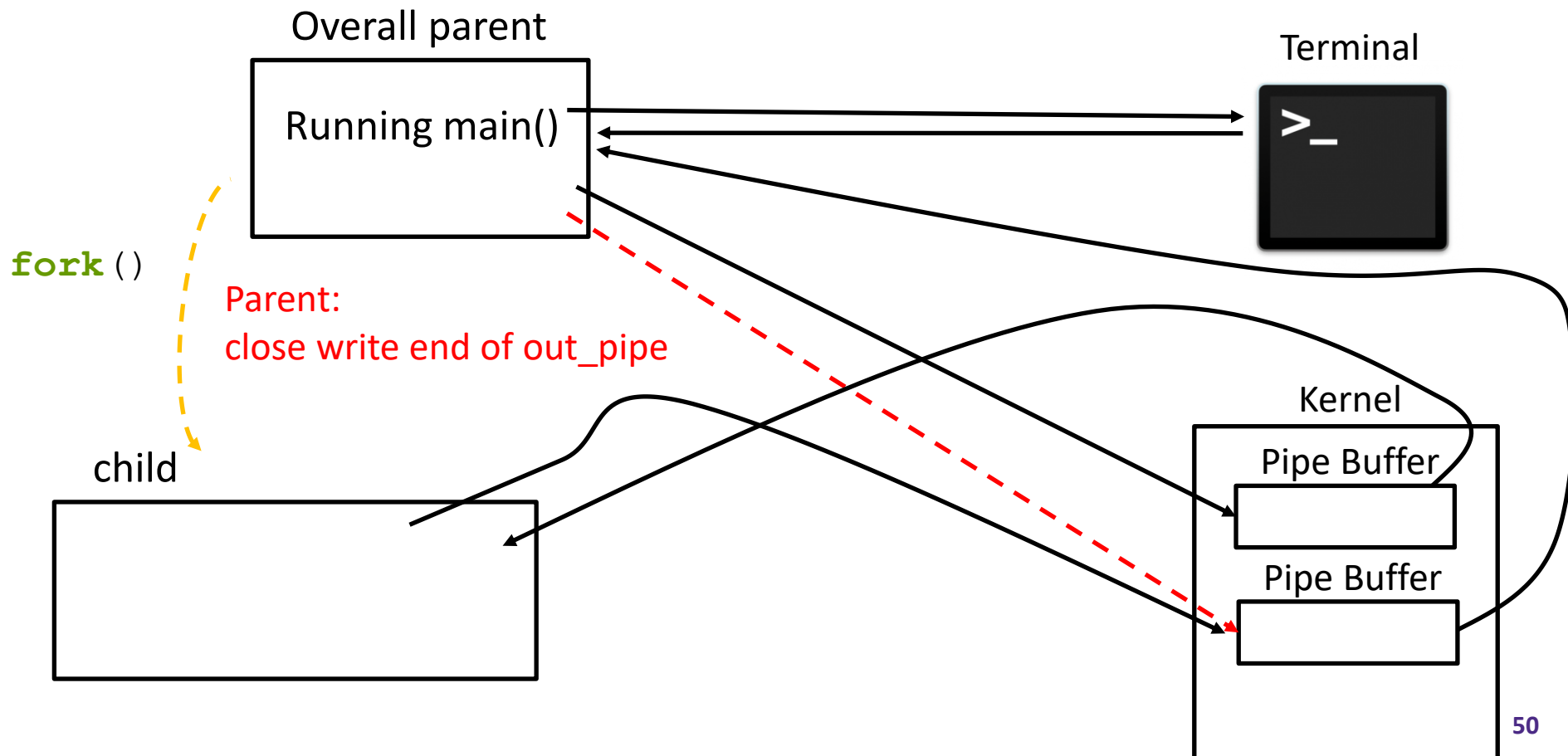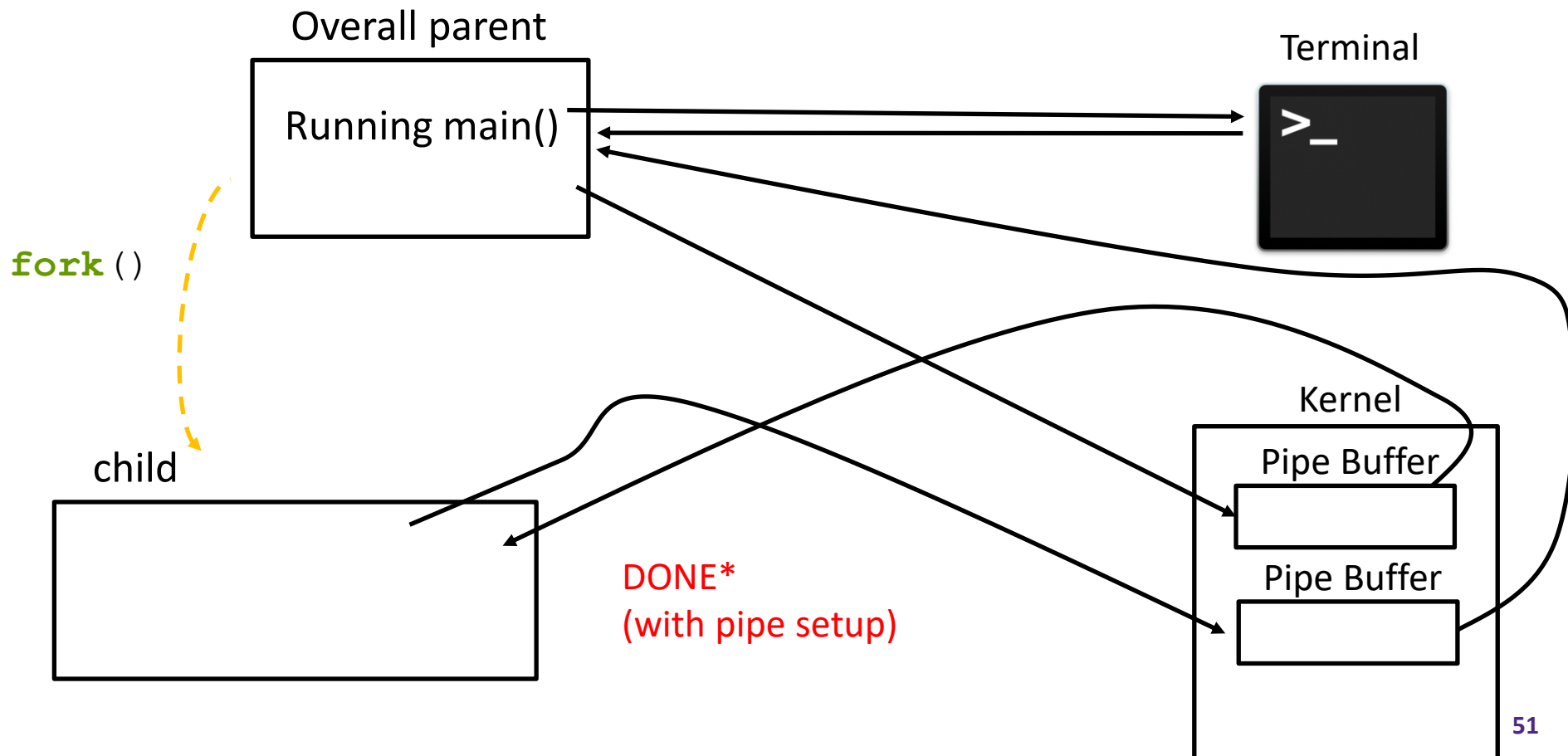   BUT send autograder input and capture output

Overall parent

Running main()

Terminal

fork()

child

execvp("./numbers", …);

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

Kernel

Pipe Buffer

Pipe Buffer

52

Lecture Outline

❖ Intro to file descriptors

❖ File Descriptors: Big Picture

❖ Redirection & Pipes

❖ **Unix Commands & Controls**

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.

❖ <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

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

cd: change directory

A. **ls > wc**

B. **cd . && ls wc**

ls: list directory contents

C. **ls | wc**

D. **ls && wc**

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

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**

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

D. **ls && wc**

E. **The correct answer is not listed**

```
ls | wc -l
```
would be preferred.

F. **We're lost…**