# Unix, HW4, and Move

## 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

- Project released
  - Due May 1st at midnight, please get started if you haven't already
- HW4
  - To be posted shortly after lecture
  - Should have everything you need after Today's Lecture
- Checkin to be released soon


pollev.com/tqm

- What is your primary OS?


pollev.com/tqm

- What do you think is the most used OS?


pollev.com/tqm

- Any questions?

# Lecture Outline

- **Brief History**
- UNIX Shell & Commands
- HW4 Demo
- Move

# Multics: The Precursor

- **Multiplexed Information and Computing Service**

- Early time-sharing operating system
  - Time sharing: the sharing of a computer (mainframe) across multiple users at the same time
  - Necessary pre – personal computers (~1975)

- Started development in 1964
  - funded in part by Bell labs
- Bell Labs pulls out of Multics in 1969

# "Unics"

- Ken Thompson and Dennis Ritchie lead the development of Unix
  - Both worked on Multics under Bell Labs

- Took some inspiration from Multics
  - Hierarchical file system
  - Text command line shell
  - The name:
    - Multics: Multiplexed Information and Computing Service
    - Unics: Uniplexed Information and Computing Service
    - At some point "Unics" became "Unix“
  - Unix rejected the overcomplexity of Multics

# UNIX

- Originally (1970) was a singletasking system, without name or backing, and written in PDP assembly
- Functionality and multitasking added as other departments in Bell Labs needed them
- Departments kept adopting UNIX instead of built in OS's.
  - As a result, a support team was created, a UNIX Programmer's Manual was written, and man pages were created

# UNIX and C


- B programming language by Ken Thompson
  - Was intended for writing UNIX utilities
- Dennis Ritchie modified B to make New B
  - Added things like types! (int, char, etc.)
- More features were added to New B, heavily influenced by its use in UNIX
- UNIX was soon re-written in C
  - One of the first operating systems (re)written in a higher-level-language (aka, not assembly)

# Unix Adoption

- 1973: Unix was first presented formally outside of Bell Labs. Leading to many requests for the system

- Due to a 1956 decree, Bell System could not turn UNIX into a commercial product.
  - Bell had to license the product to anyone who asked
  - Code was “open source” of sorts.

- UNIX was continually updated, and C was as well.
  - Included the addition of pipes and other features
  - These updates made UNIX more portable to other systems.

# UNIX Design Philosophy

- Philosophy behind development of UNIX that spread to standards for developing software generally.
  - Arguable more influential than UNIX itself

- Short version:
  - 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.

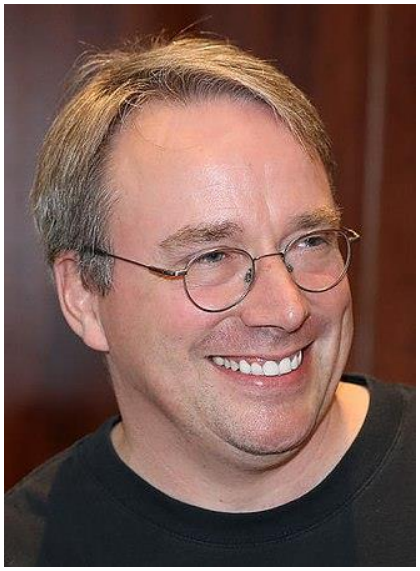
- Extra short version: "Keep it Simple, Stupid."

# GNU

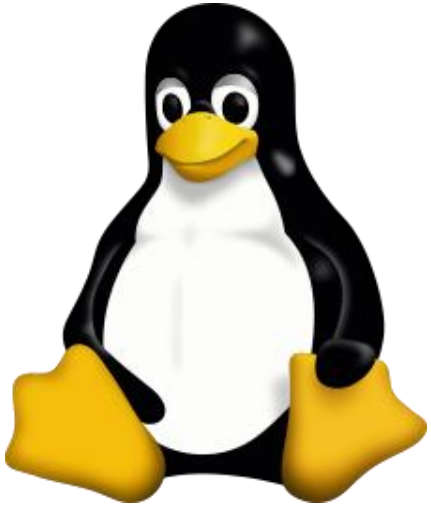
- In 1983, Bell Systems split up due to anti-trust laws.
  - A successor (AT&T) then turned UNIX into a commercial product, limiting rights to distribute/change/adapt/etc. UNIX

- Later that year, GNU is founded by Richard Stallman
  - GNU Not Unix
  - Copyleft
  - Goal: create a complete UNIX compatible system composed entirely of free software
  - Developed many required programs (libraries, editors, shell, compilers …) but missing low level elements like the kernel

# Linux



- By 1991, a UNIX-like kernel that was Free Software did not exist
- Linus Torvalds was studying operating systems and wrote his own called Linux
  - This would be published under GPL 2 (GNU Public License)
- Blew up in popularity due to being free and open source



# Unix-Like

- Almost all operating systems are UNIX related
  - "Genetically" related with historical connection to the original code base
  - Through the UNIX trademark once a system meets the Single UNIX Specification and is certified
  - Through "functionally" being UNIX-like. Behaving in a manner that is consistent with UNIX design and specification
    - Linux falls under this one

- Most Operating systems are Unix Like
  - Linux, macOS, iOS, Chrome OS, Android, etc.

# Lecture Outline

- Brief History
- **UNIX Shell & Commands**
- HW4 Demo
- Move

# 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 > file`, redirects the stdout of a command to be written to the specified file
- 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

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

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.

# 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`"

# Lecture Outline

- Brief History
- UNIX Shell & Commands
- **HW4 Demo**
- Move

# 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:

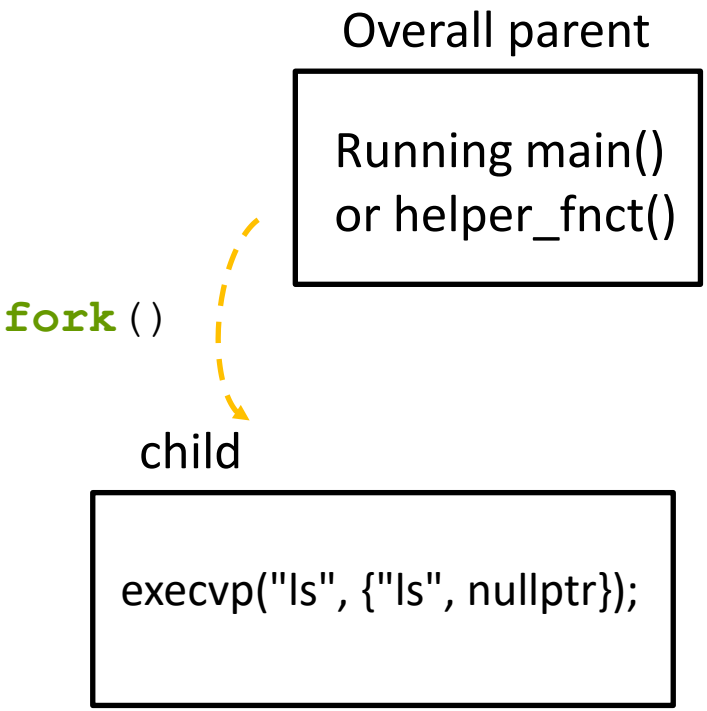
```
./solution_binaries/pipe_shell
```

# 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"`

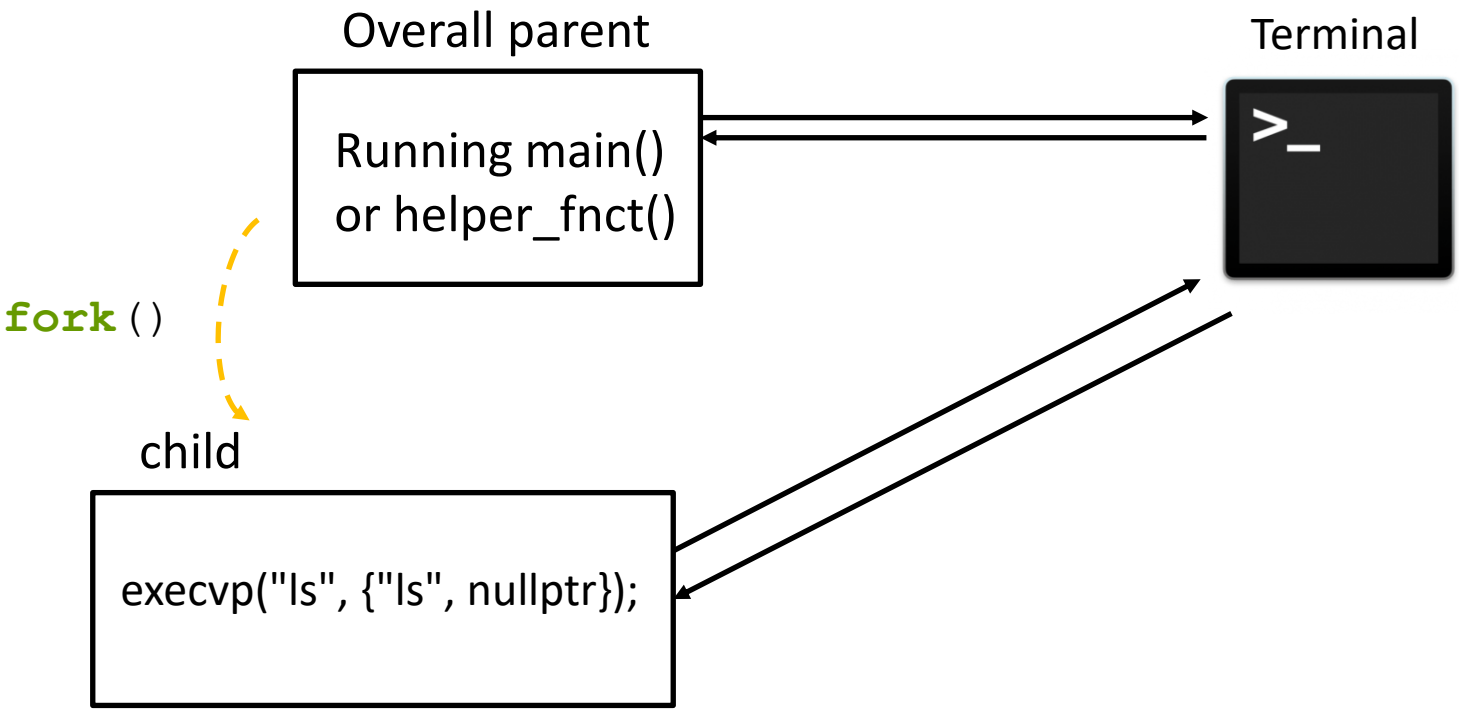
# HW4 Example Line

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



# HW4 Example Line

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



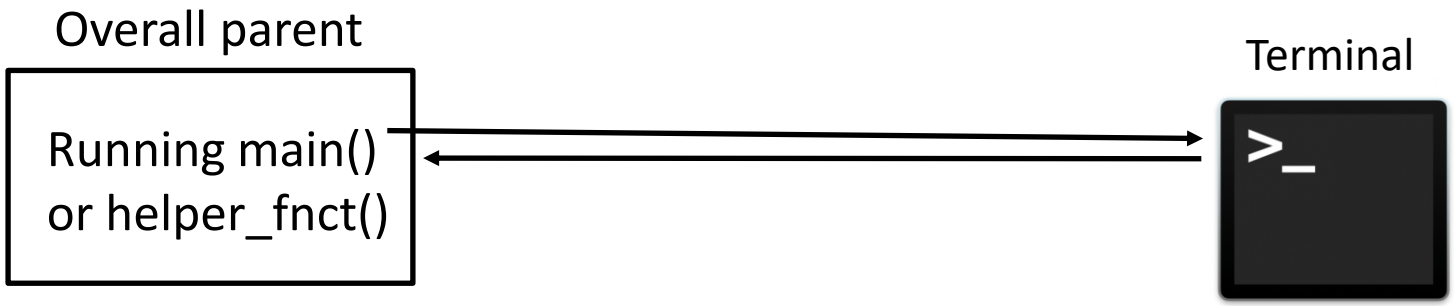
# HW4 Hints

- 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

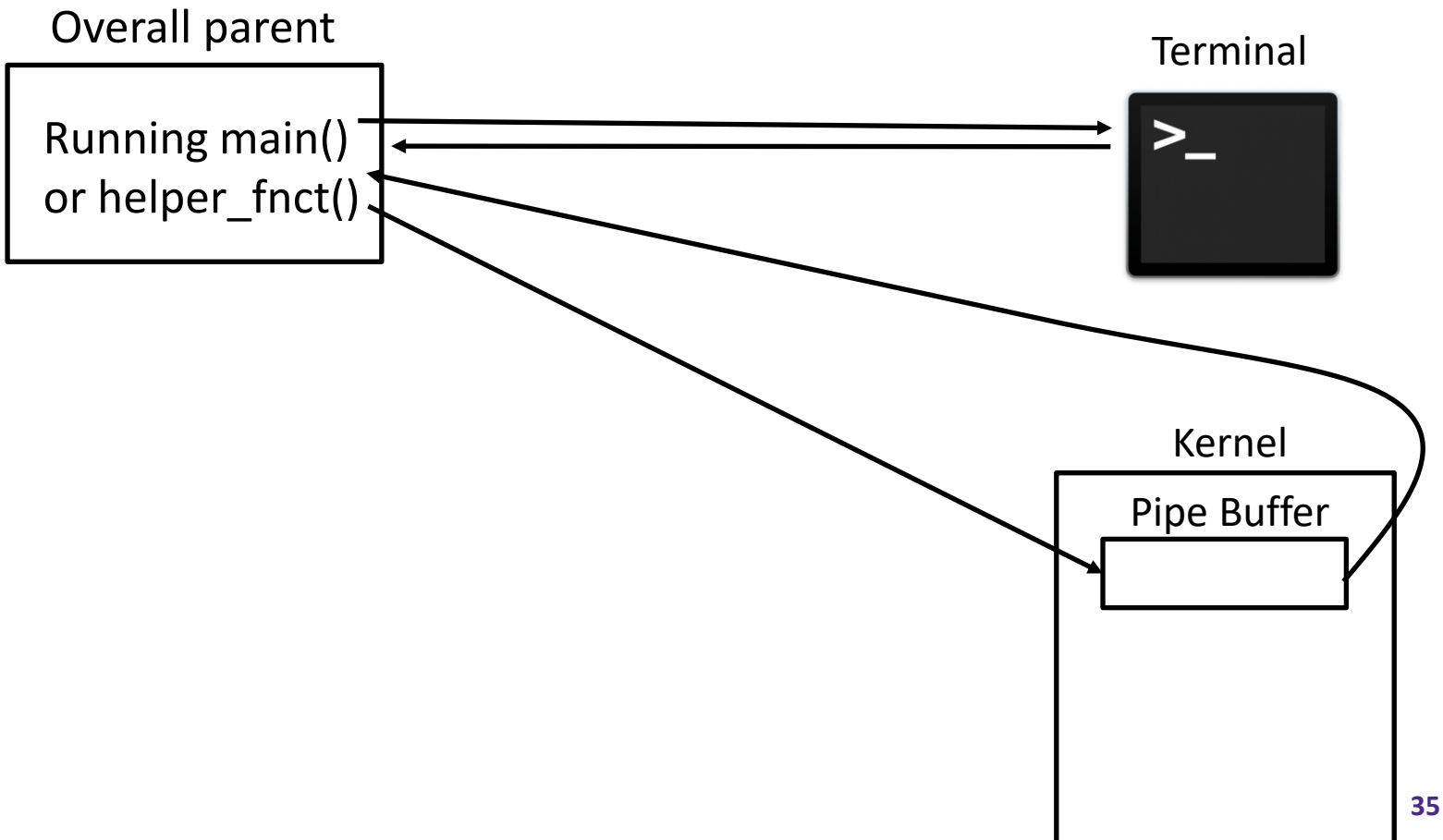
- Consider the case when a user inputs
  - `"ls | wc"`




Kernel

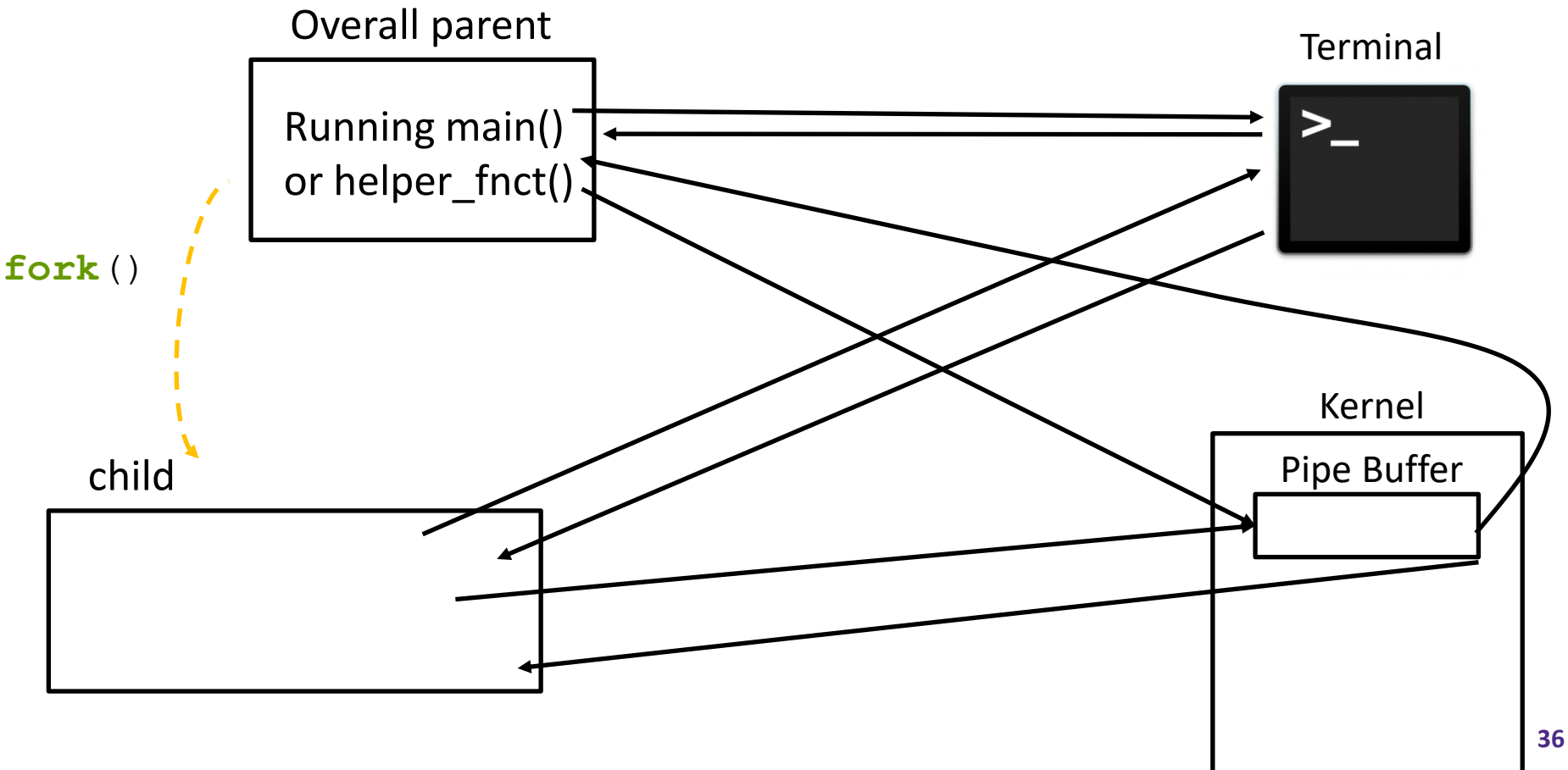
# HW4 Example Line 1

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



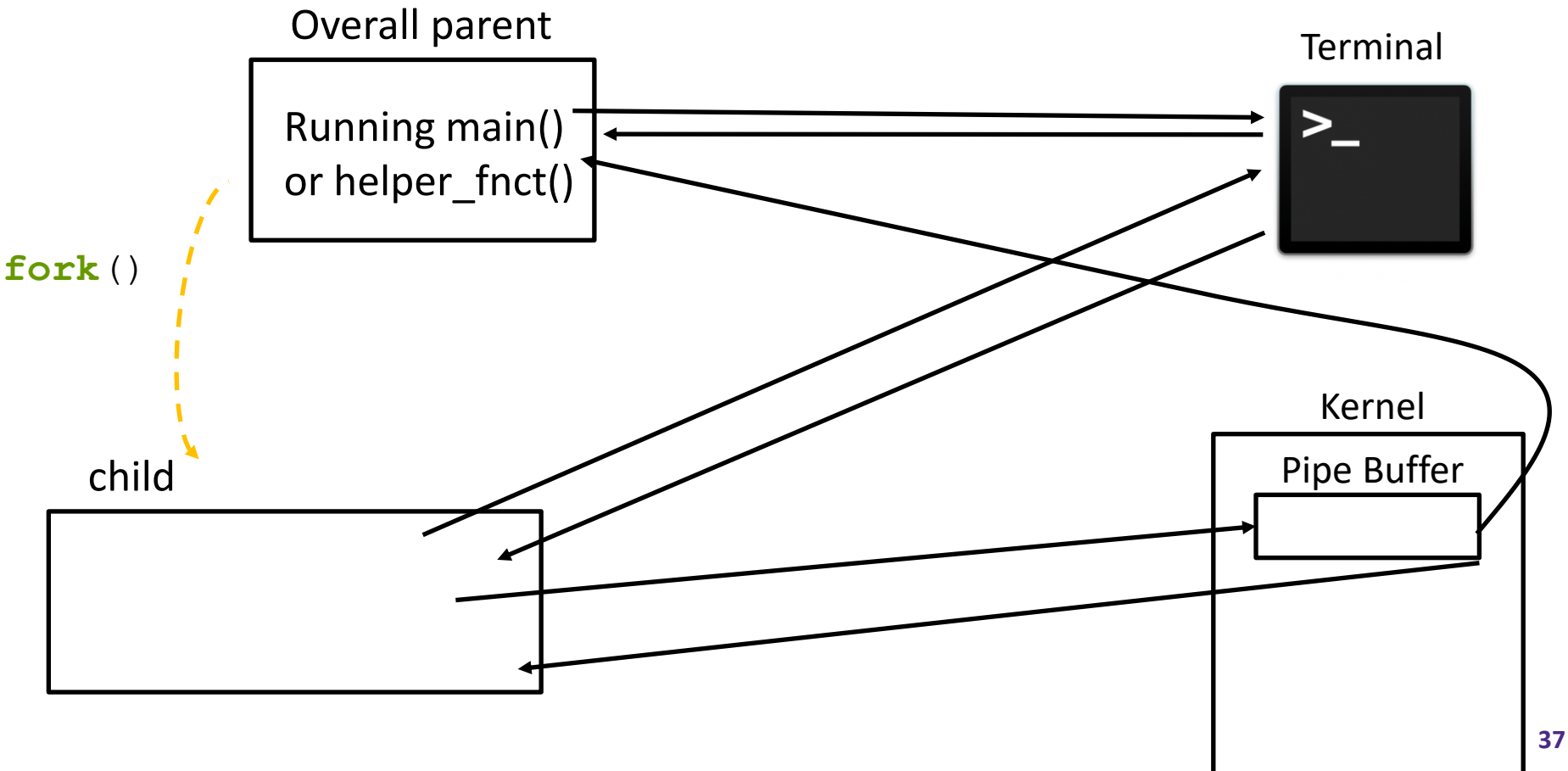
# HW4 Example Line 1

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



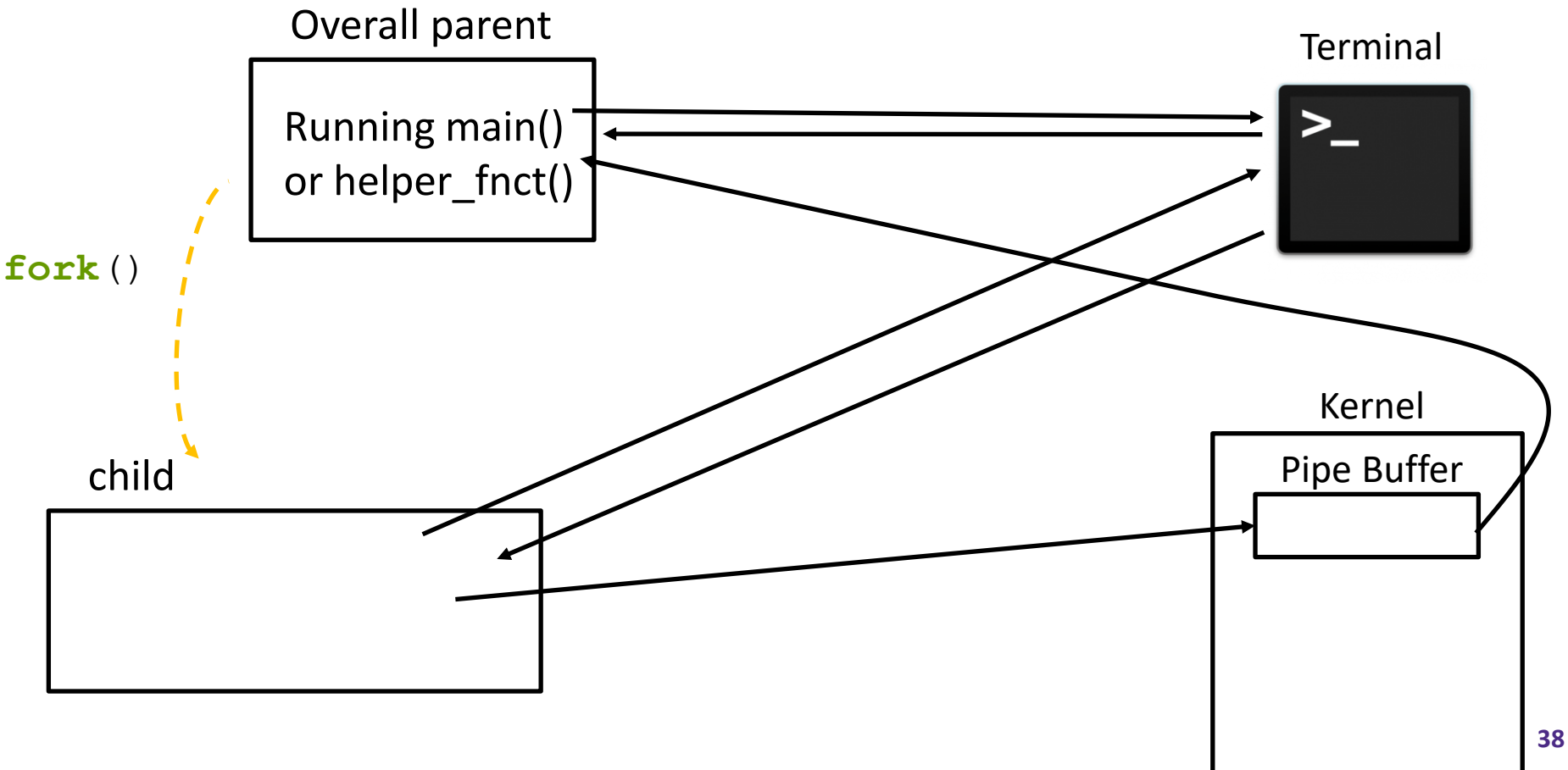
# HW4 Example Line 1

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



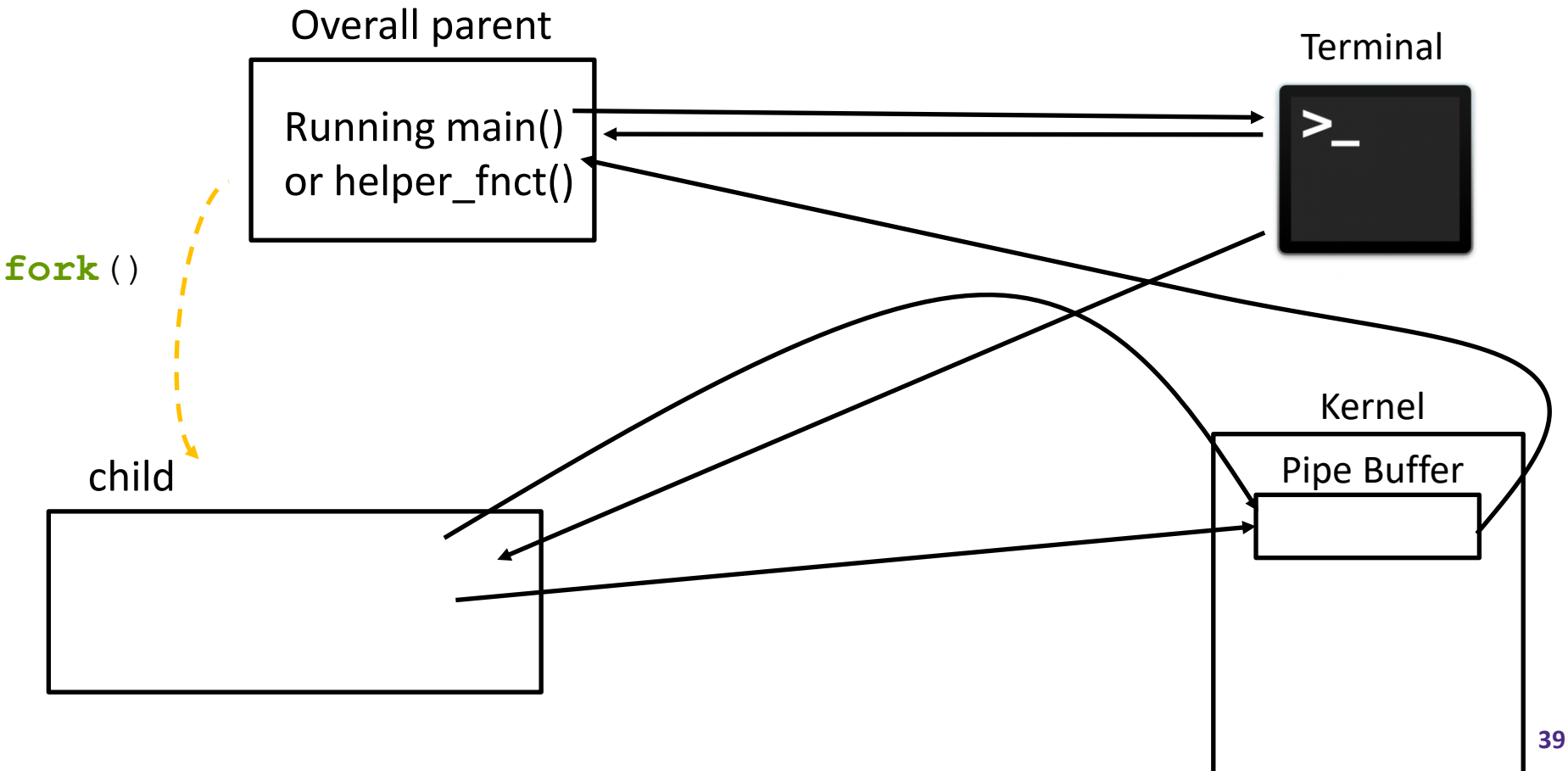
# HW4 Example Line 1

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



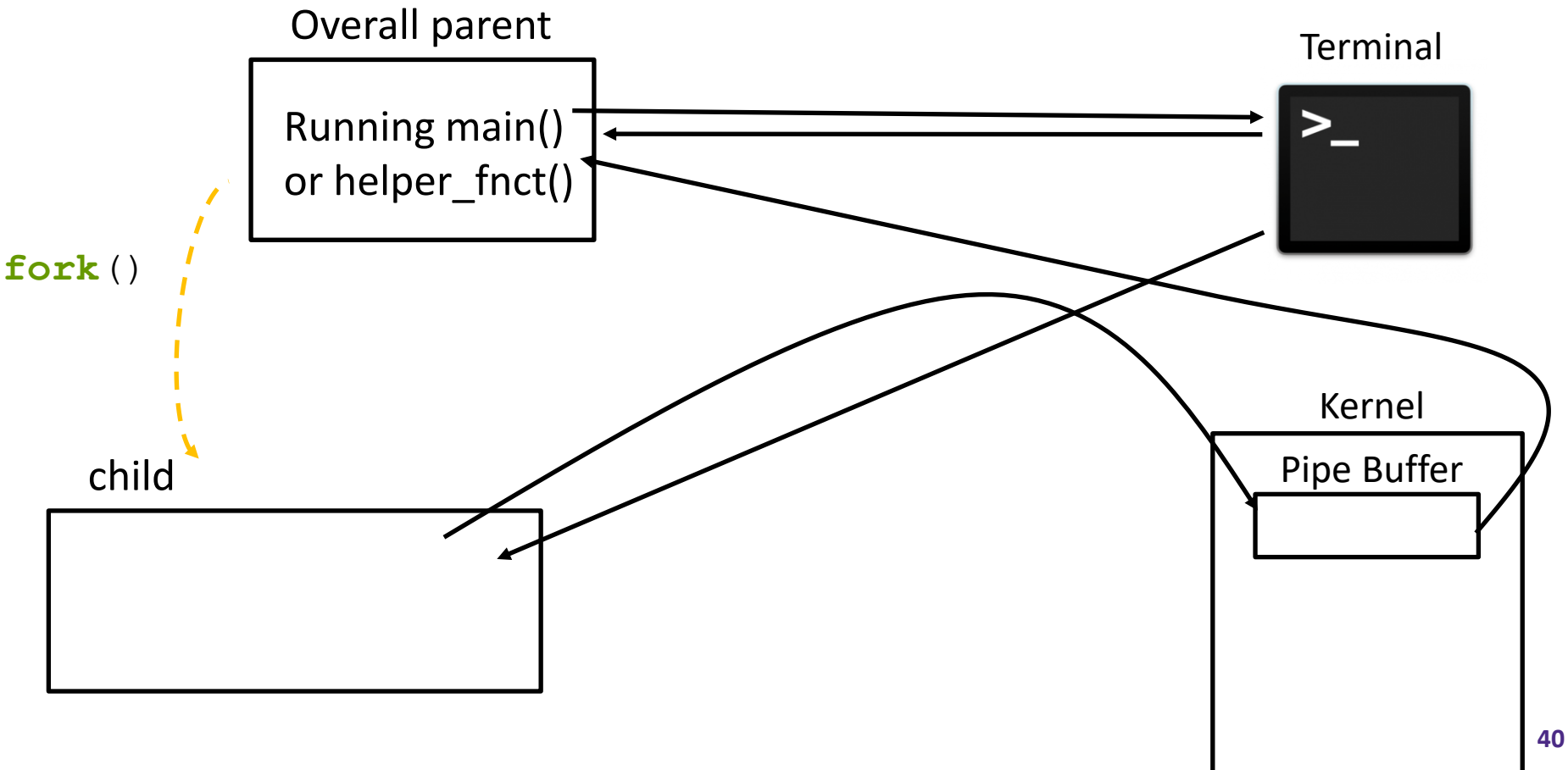
# HW4 Example Line 1

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



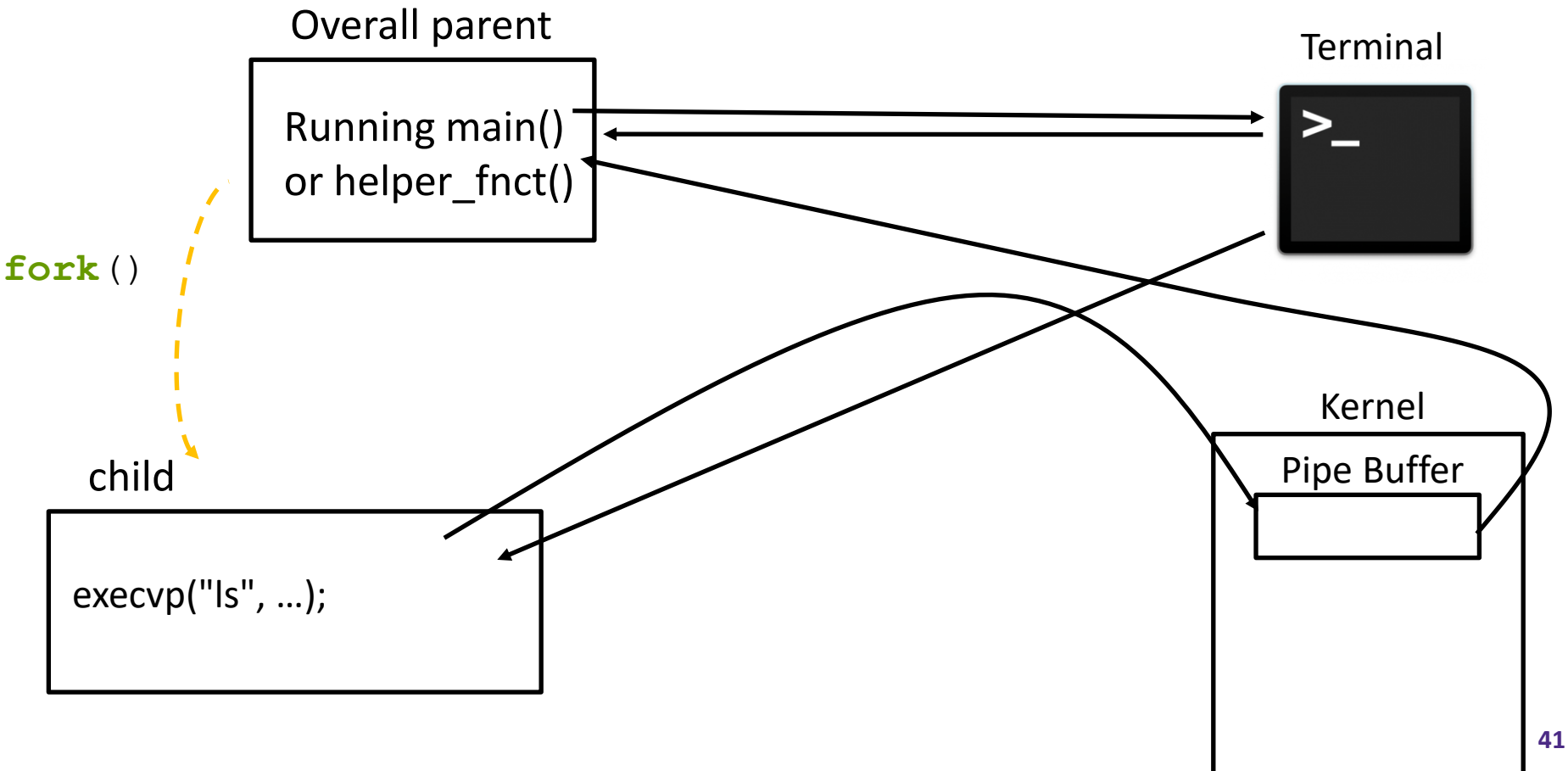
# HW4 Example Line 1

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



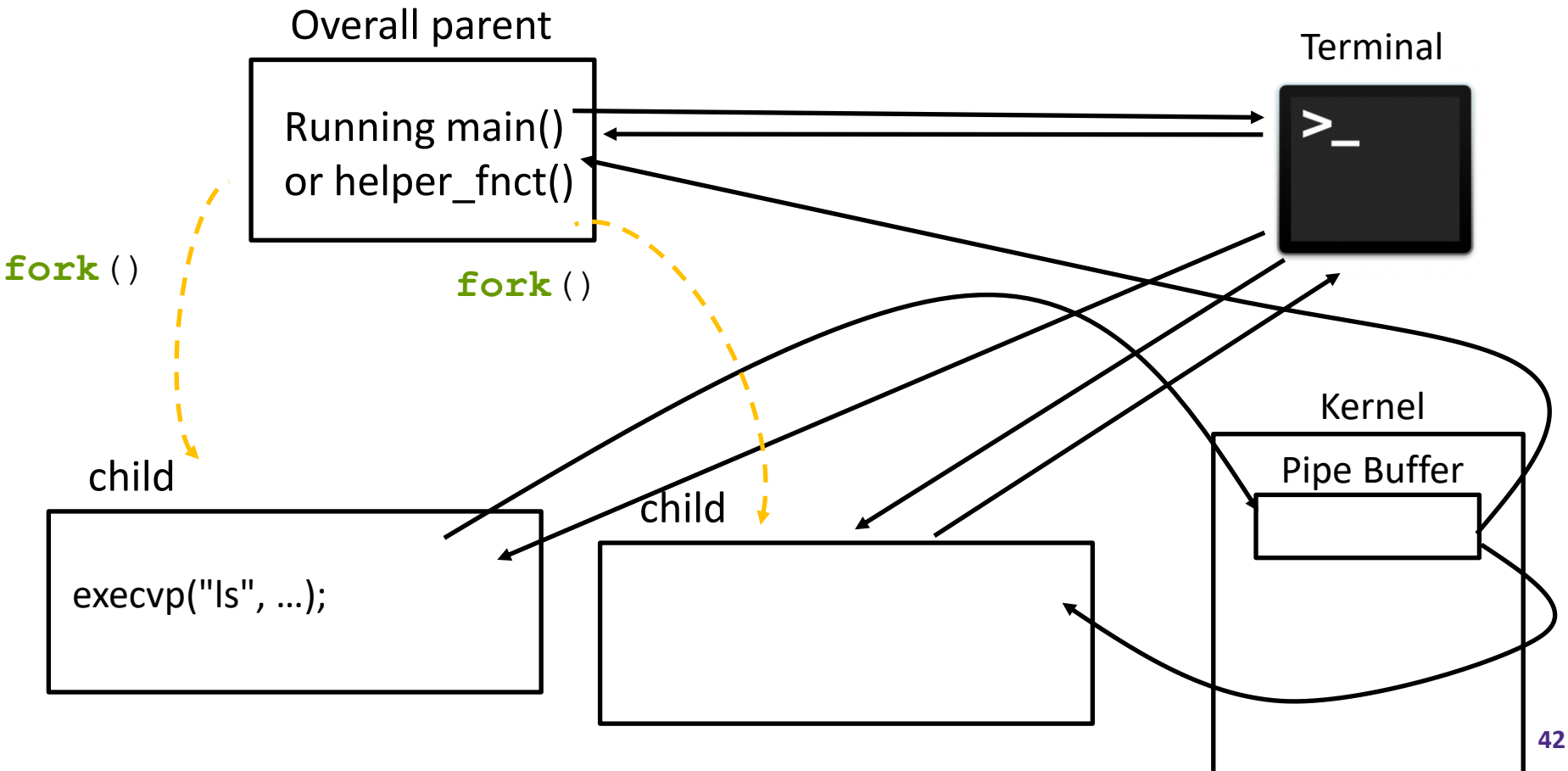
# HW4 Example Line 1

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



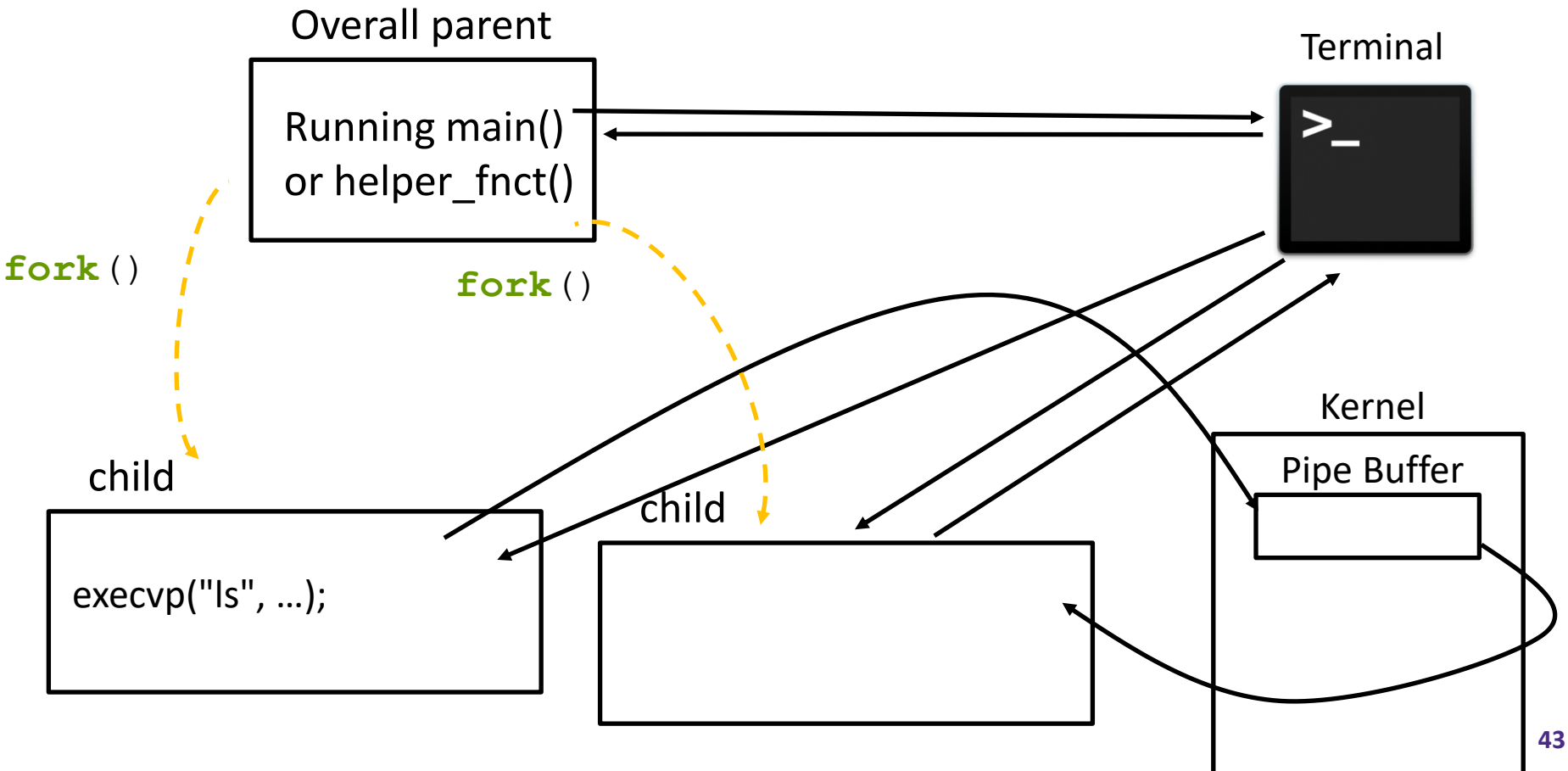
# HW4 Example Line 1

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



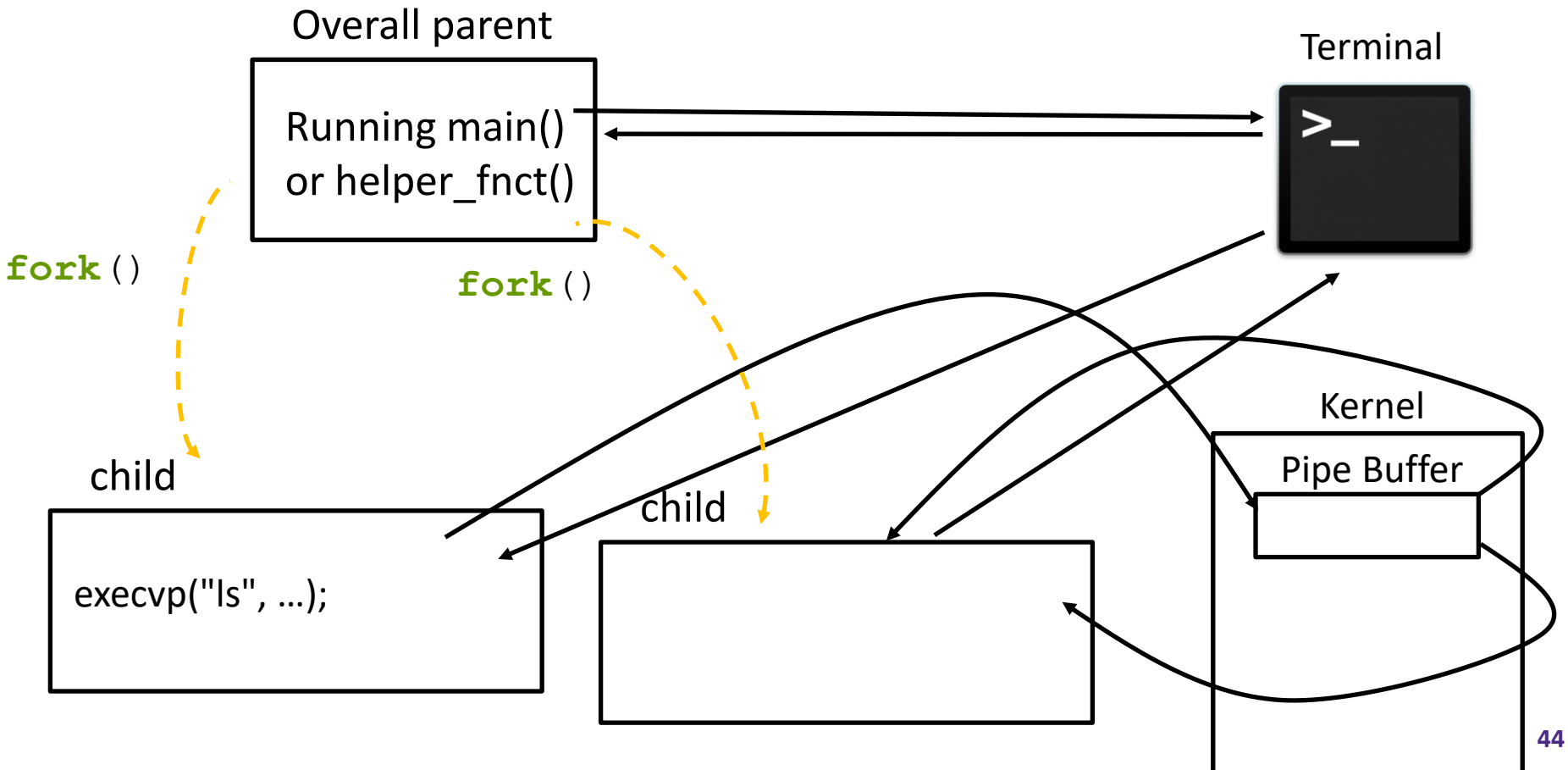
# HW4 Example Line 1

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



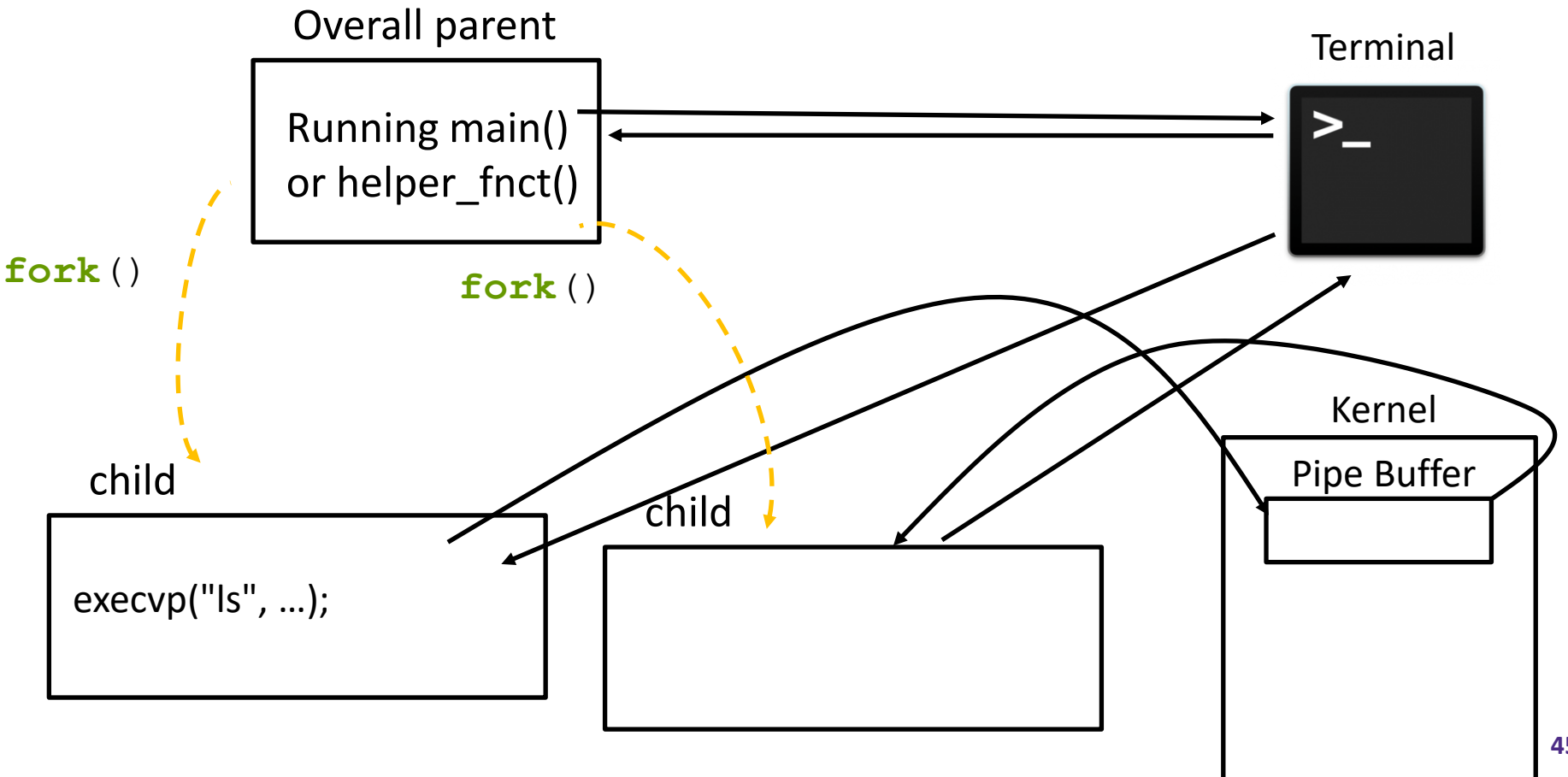
# HW4 Example Line 1

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



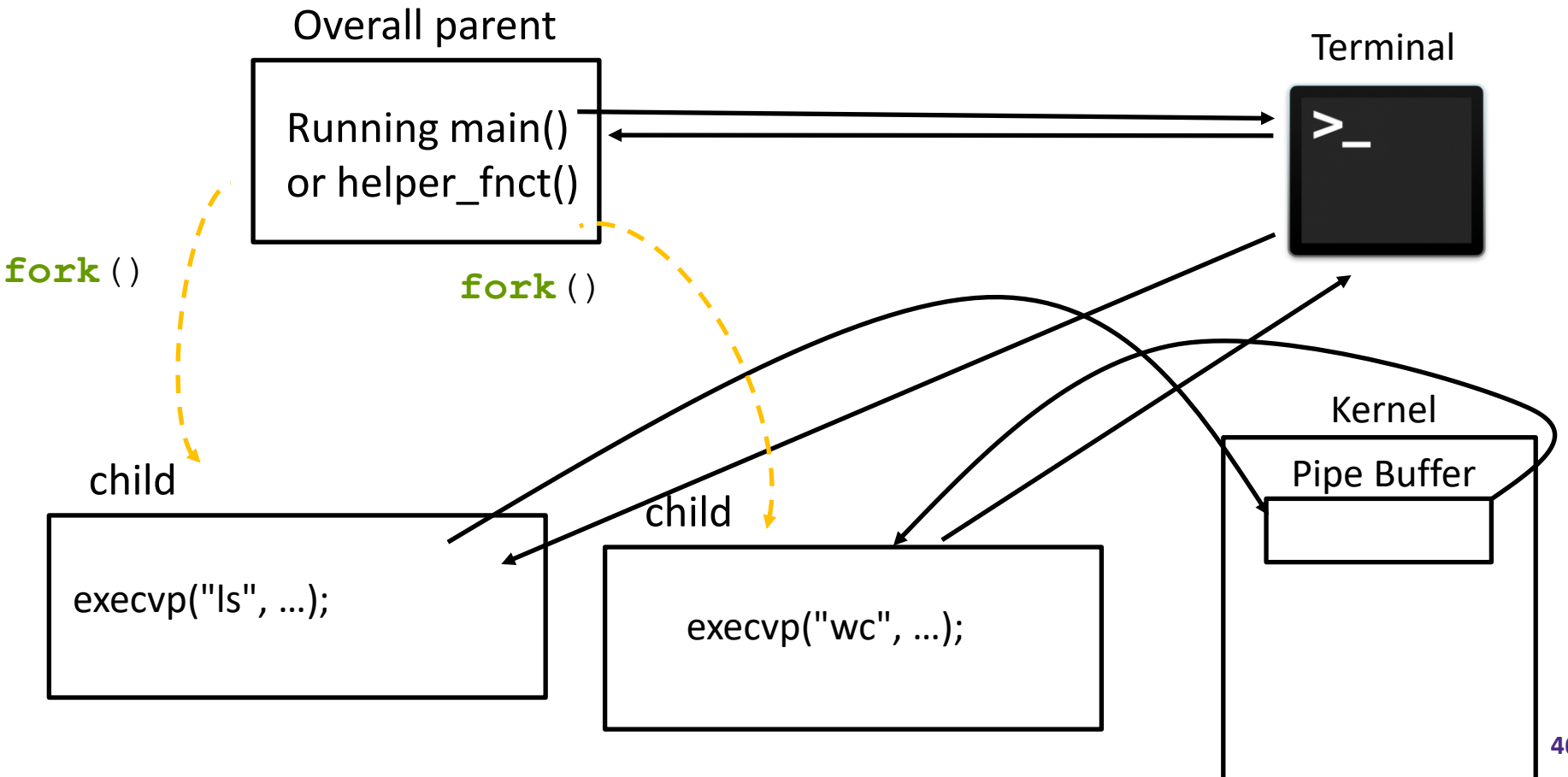
# HW4 Example Line 1

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



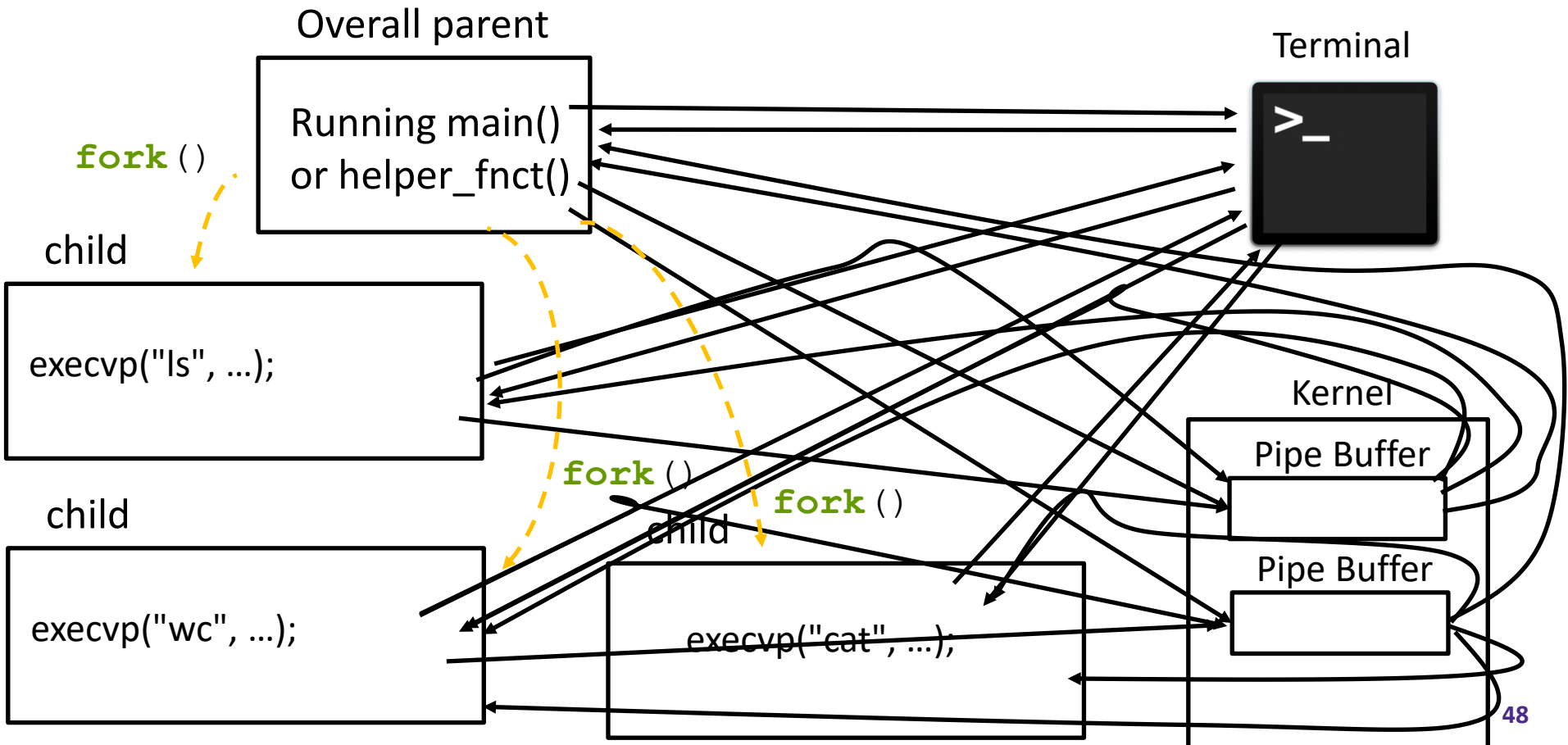
# HW4 Example Line 1

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



# HW4 Example Line 2

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



# Lecture Outline

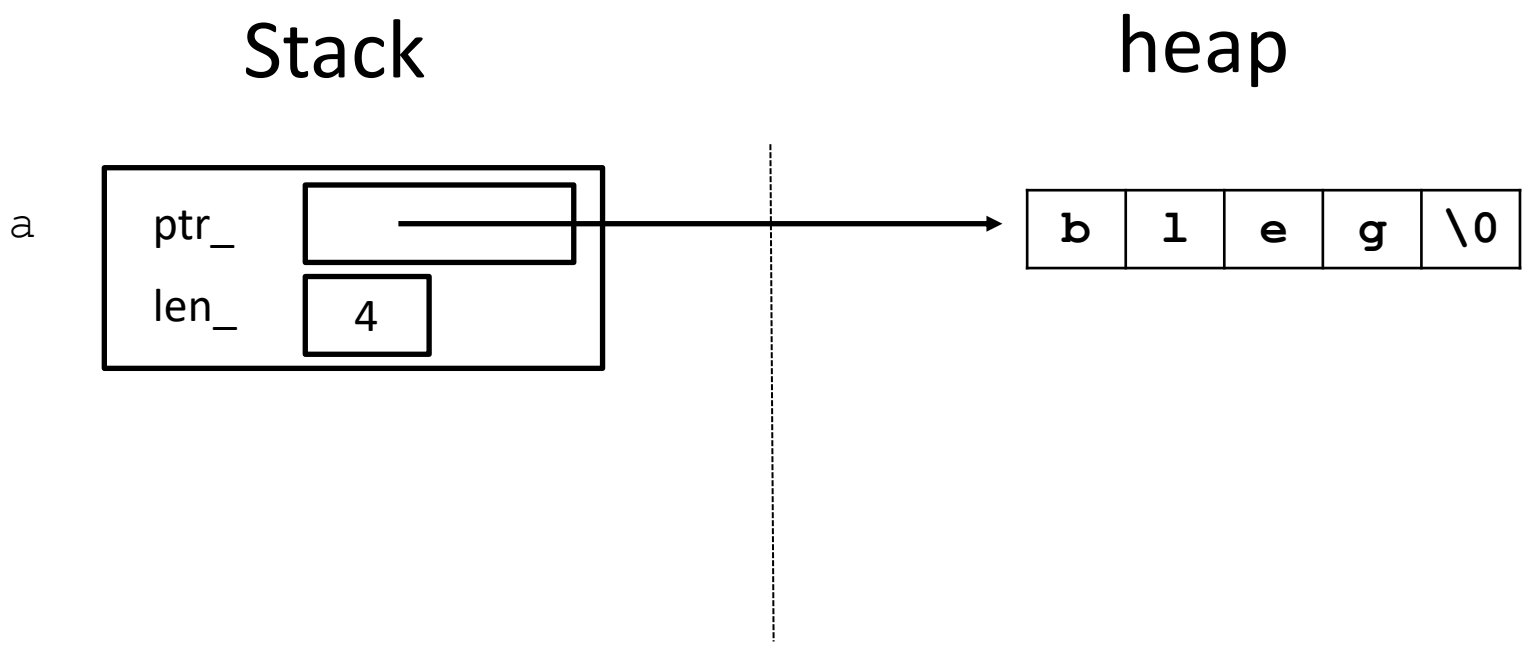
- Brief History
- UNIX Shell & Commands
- HW4 Demo
- **Move**

# Copy Semantics: close up look

- Internally a string manages a heap allocated C string and looks something like:

```
int main(int argc, char **argv) {
  std::string a{"bleg"};


}
```



# Copy Semantics: close up look

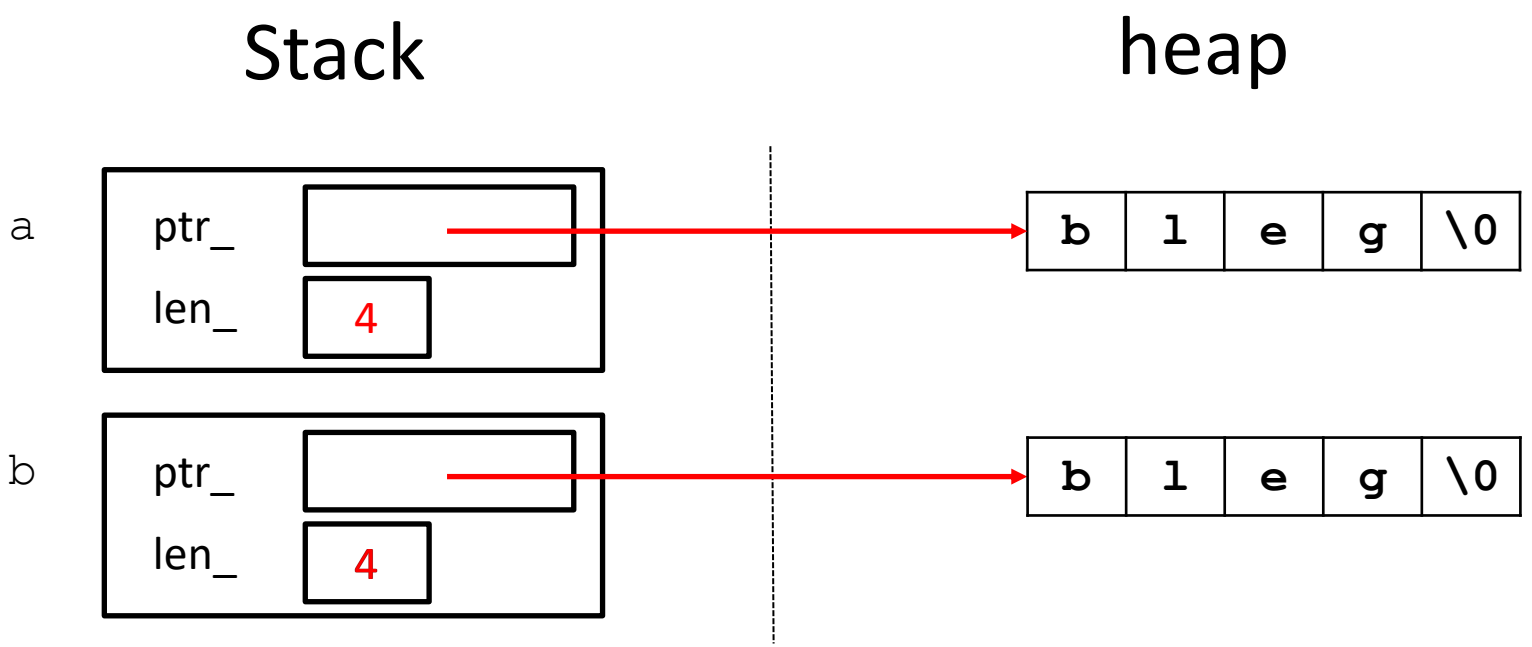
- When we copy construct string **b**

```
int main(int argc, char **argv) {
  std::string a{"bleg"};

  std::string b{a};
}
```

we could get something like:

This is another memory allocation, and we need to copy over the characters of the string



# Move Semantics (C++11)

- "Move semantics" move values from one object to another without copying ("stealing")
  - A complex topic that uses things called "*rvalue references*"
    - Mostly beyond the scope of this class

```
int main(int argc, char **argv) {
  std::string a{"bleg"};

  // moves a to b
  std::string b{std::move(a)};
  std::cout << "a: " << a << std::endl;
  std::cout << "b: " << b << std::endl;


  return EXIT_SUCCESS;
}
```

a: ""
b: "bleg"

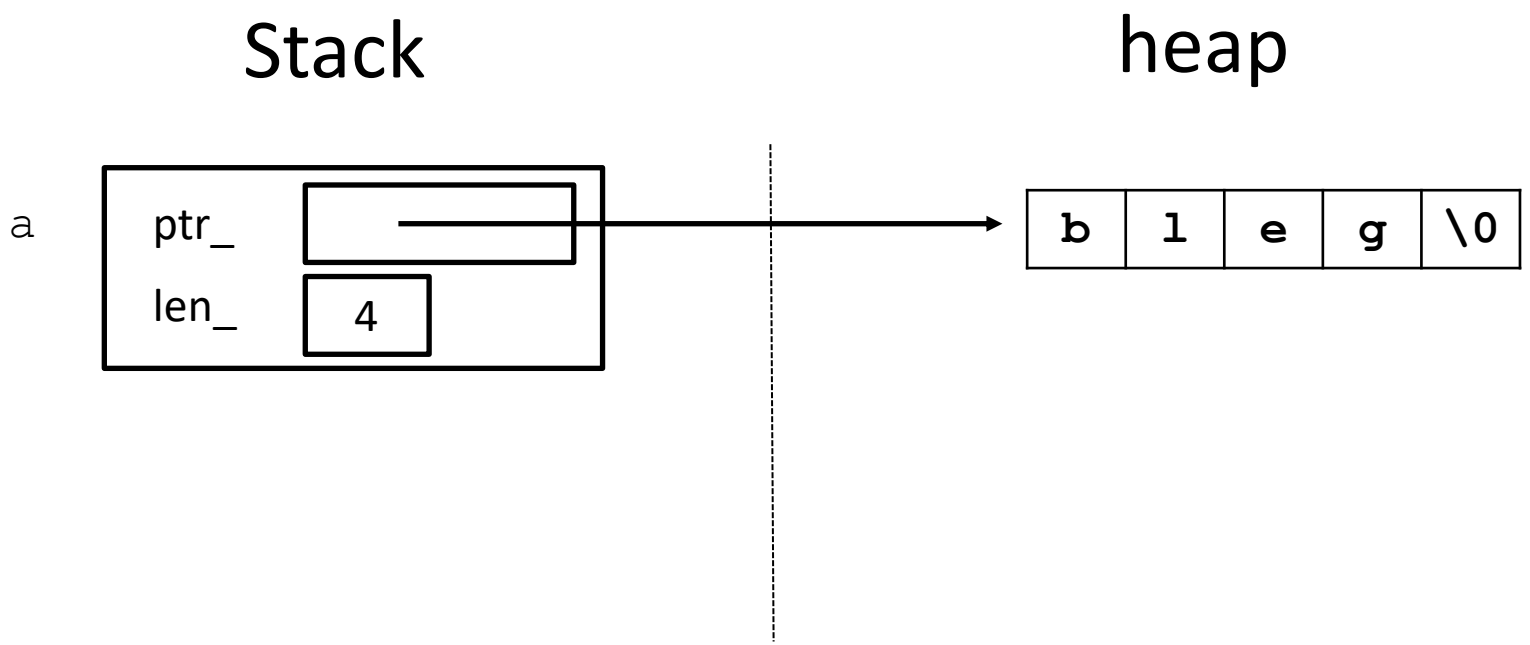
Note: we should NOT access 'a' after we move it. It is undefined to do so, it just so happens it is set to the empty string

# Move Semantics: close up look

- Internally a string manages a heap allocated C string and looks something like:

```
int main(int argc, char **argv) {
  std::string a{"bleg"};


}
```



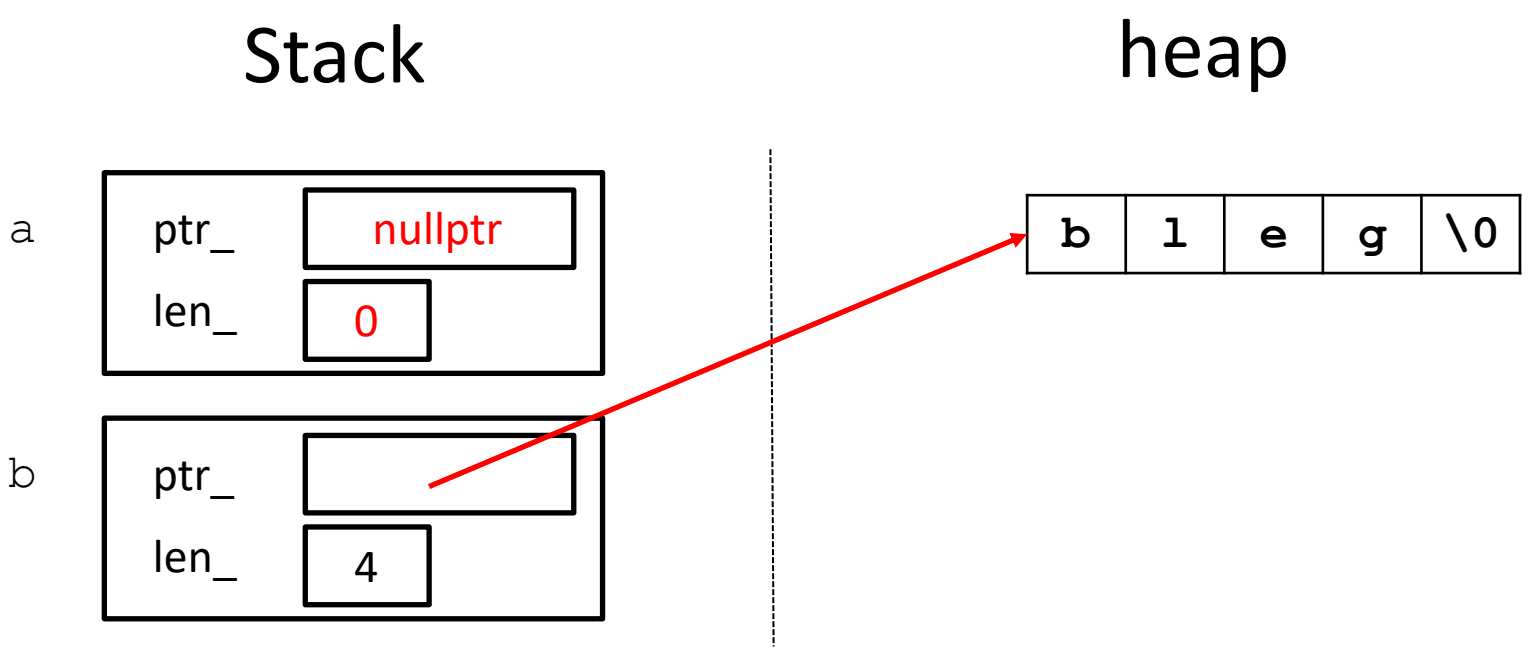
# Move Semantics: close up look

- When we use move to construct string **b**

```
int main(int argc, char **argv) {
  std::string a{"bleg"};

  std::string b{std::move(a)};
}
```

we could get something like:



# Move Semantics: Use Cases

- Useful for optimizing away temporary copies
- Preferred in cases where copying may be expensive
  - Consider we had a vector of strings… we could transfer ownership of memory to avoid copying the vector and each string inside of it.
- Can be used to help enforce uniqueness

- Rust is a systems programming language that is gaining popularity and by default it will move variables instead of copy them.

# Move Semantics: Details

- Implement a "Move Constructor" with something like:

```
Point::Point(Point&& other) {
  // ...
}
```

- Implement a "Move assignment" with something like:

```
Point& Point::operator=(Point&& rhs) {
  // ...
}
```

# Move Semantics: Details

- "Move Constructor" example for a fake **`String`** class:

```
String::String(String&& other) {
  this->len_ = other.len_;
  this->ptr_ = other.ptr_;

  other.len_ = 0;
  other.ptr_ = nullptr;
}
```

# std::move

- Use `std::move` to indicate that you want to move something and not copy it

```
Point p {3, 2};             // constructor
Point a {p};                // copy constructor

Point b {std::move(p)}; // move constructor
```