HW4 cont. & Inheritance (start)

Computer Systems Programming, Spring 2023

Instructor: Travis McGaha

TAs:

Kevin Bernat
Mati Davis
Chandravaran Kunjeti
Shufan Liu
Jialin Cai
Donglun He
Heyi Liu
Eddy Yang

pollev.com/tqm

- Any questions from previous lectures?

Logistics

- HW4 Posted — Due Thursday 4/20 @ 11:59
- Project Released! — Due Wednesday 4/26 @ 11:59
- Travis has extra Office Hours from 10:15 am to 12:15 pm this Thursday 4/13

Logistics

- Final Exam Scheduling:
  - 96 hours (4 days)
  - Opens Tuesday May 2nd @ Noon
  - Closes Saturday May 6th @ noon

Lecture Outline

- **More HW4**
- Polymorphism (start)
  - Inheritance motivation & C++ Syntax
  - Polymorphism & Dynamic Dispatch

Unix Shell Control Operators: Pipe

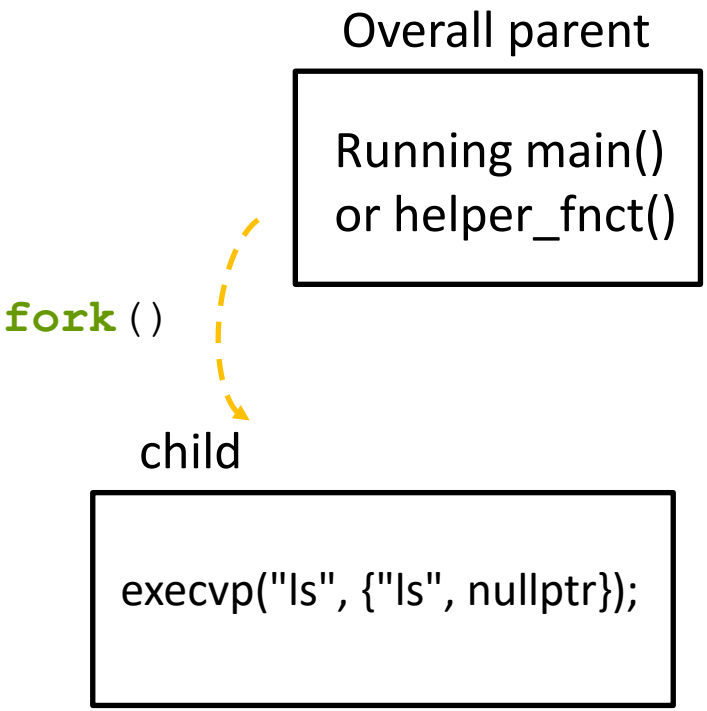
- `cmd1 | cmd2`, creates a pipe so that the stdout of cmd1 is redirected to the stdin of cmd2
  - E.g. "`history | grep valgrind`"

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"`

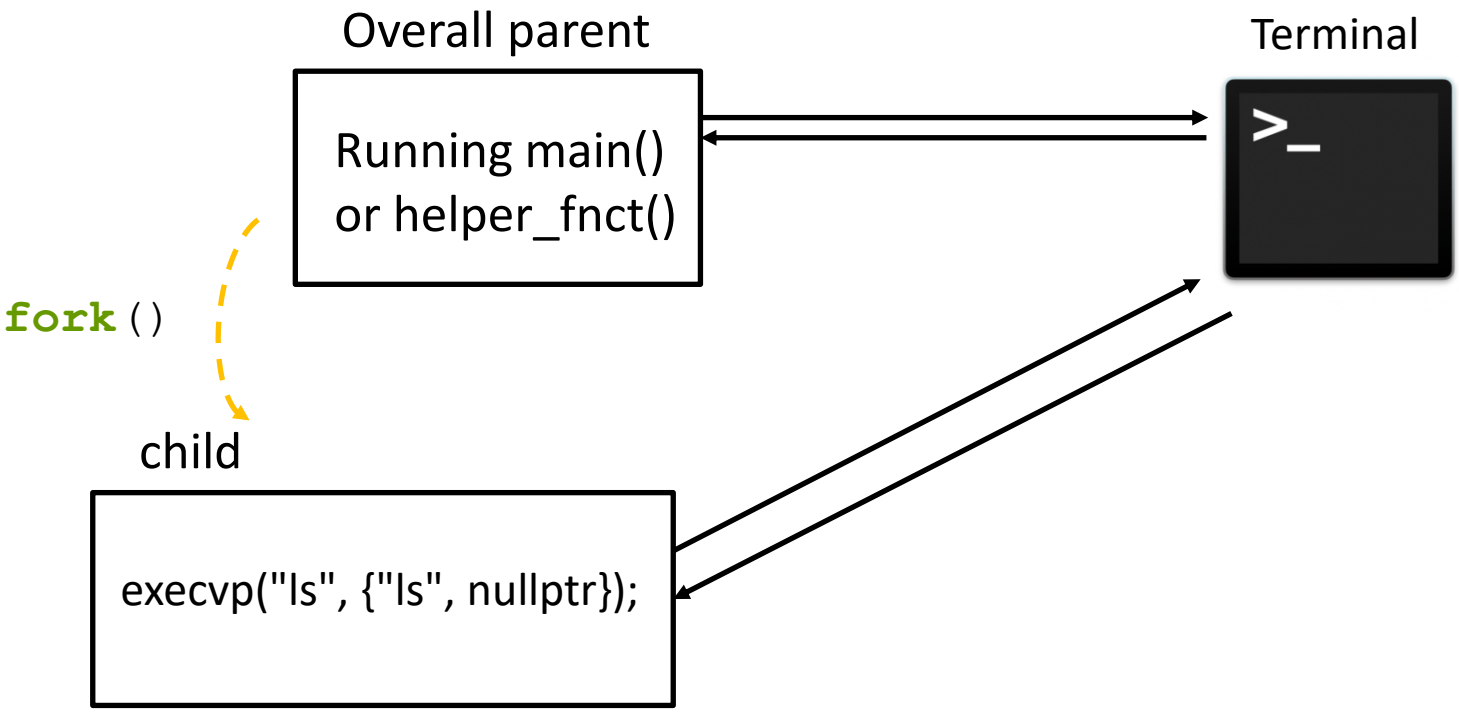
HW4 Example Line

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



HW4 Example Line

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

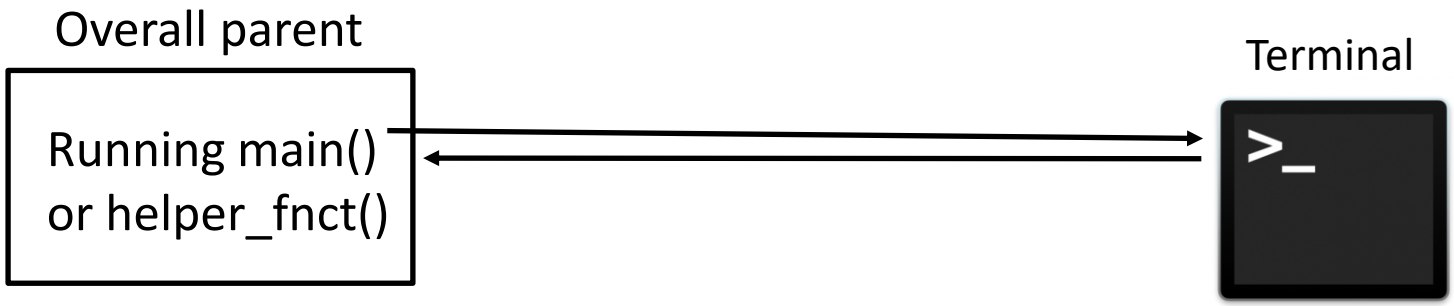


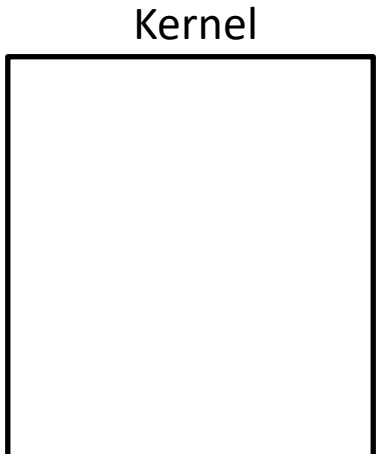
HW4 Hints

- 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

HW4 Example Line 1

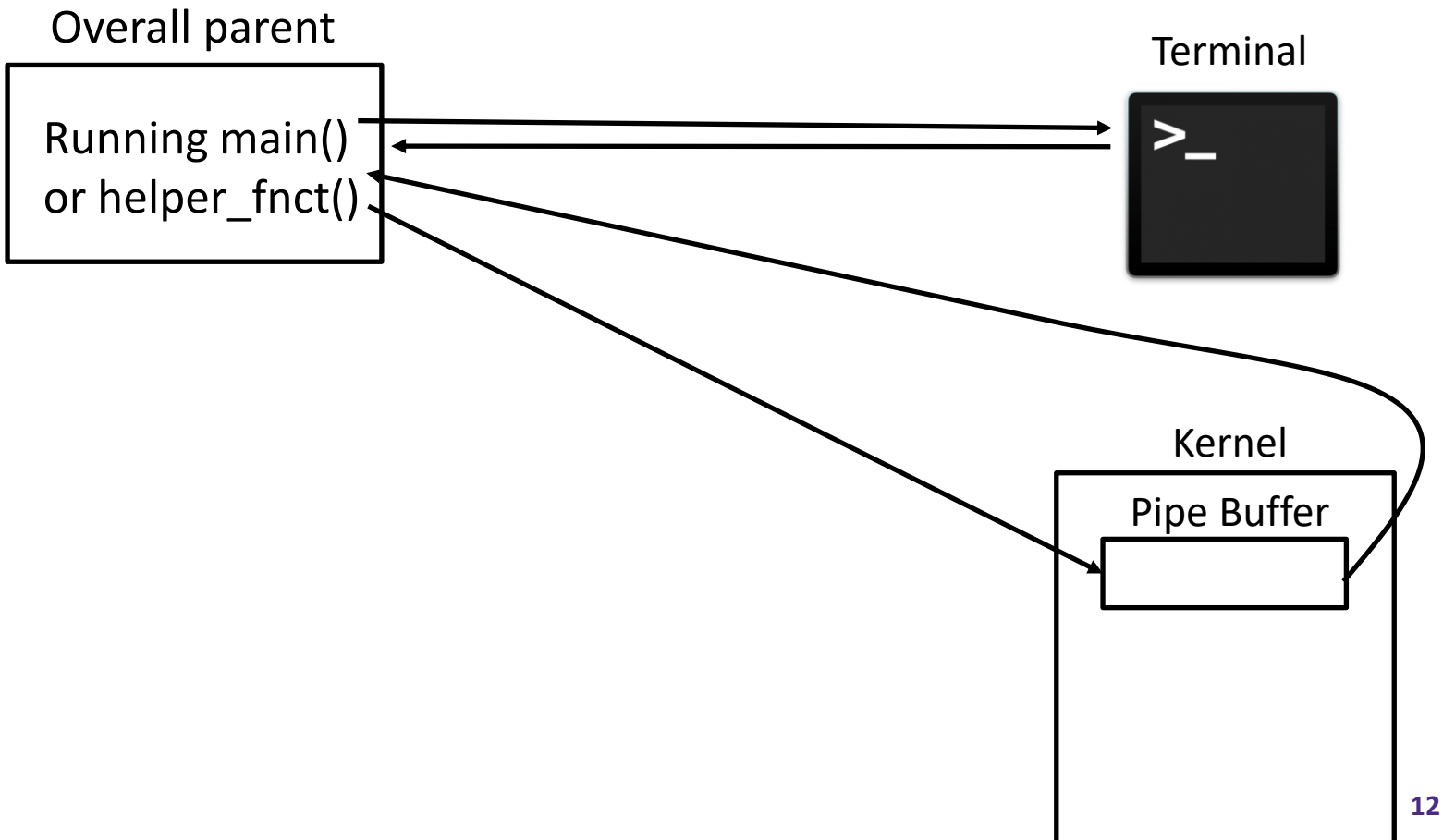
- Consider the case when a user inputs
  - `"ls | wc"`





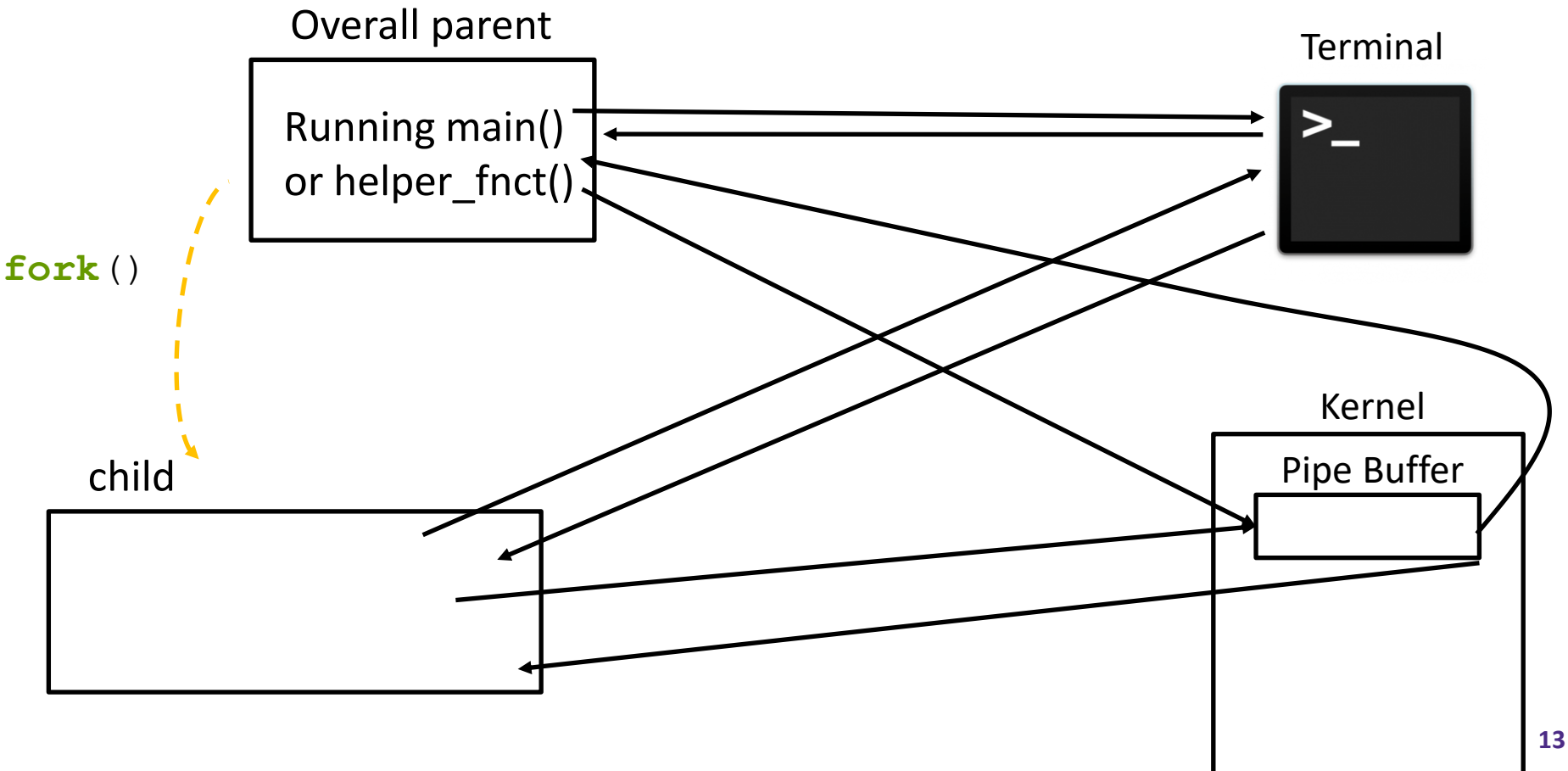
HW4 Example Line 1

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



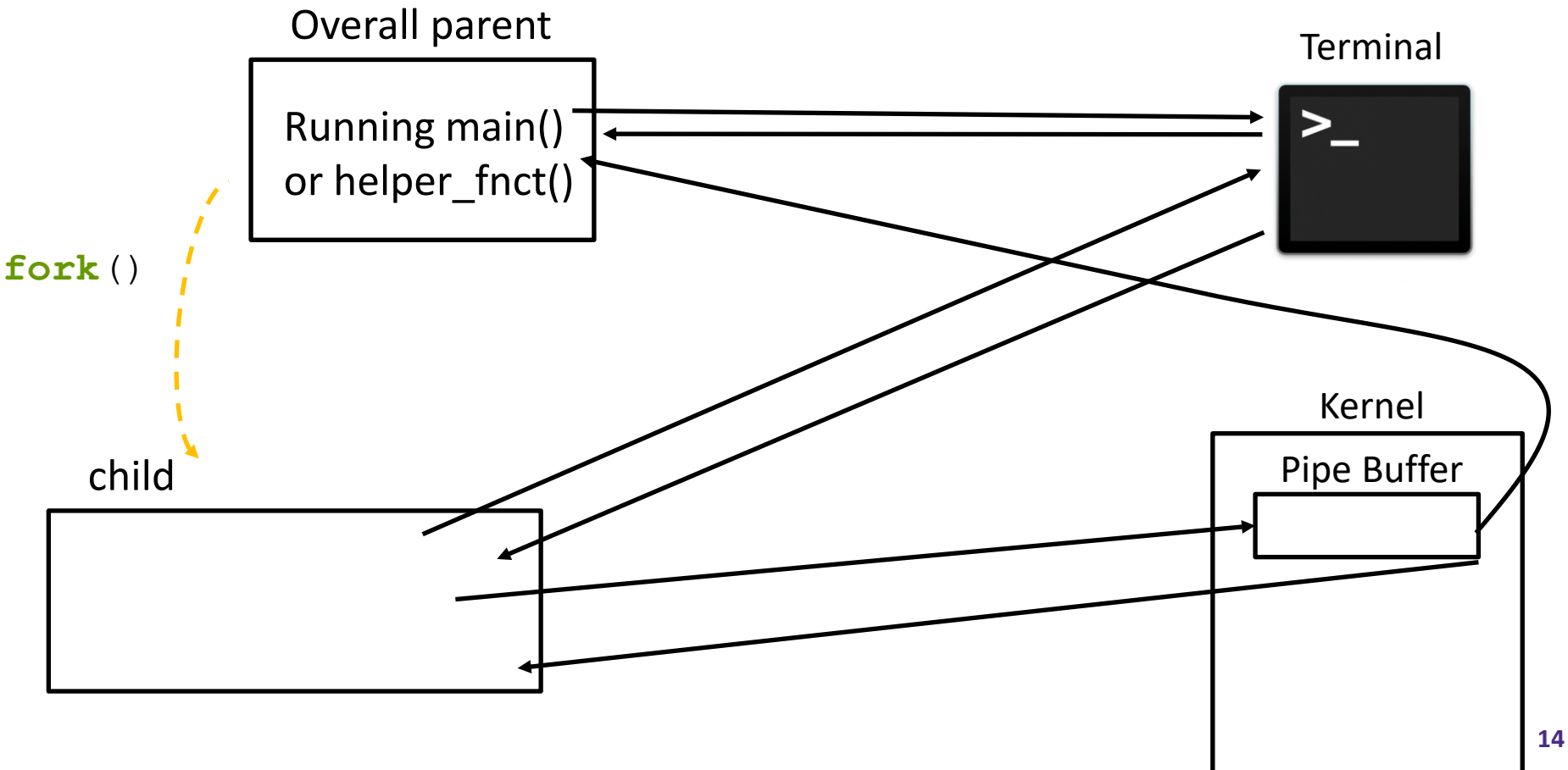
HW4 Example Line 1

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



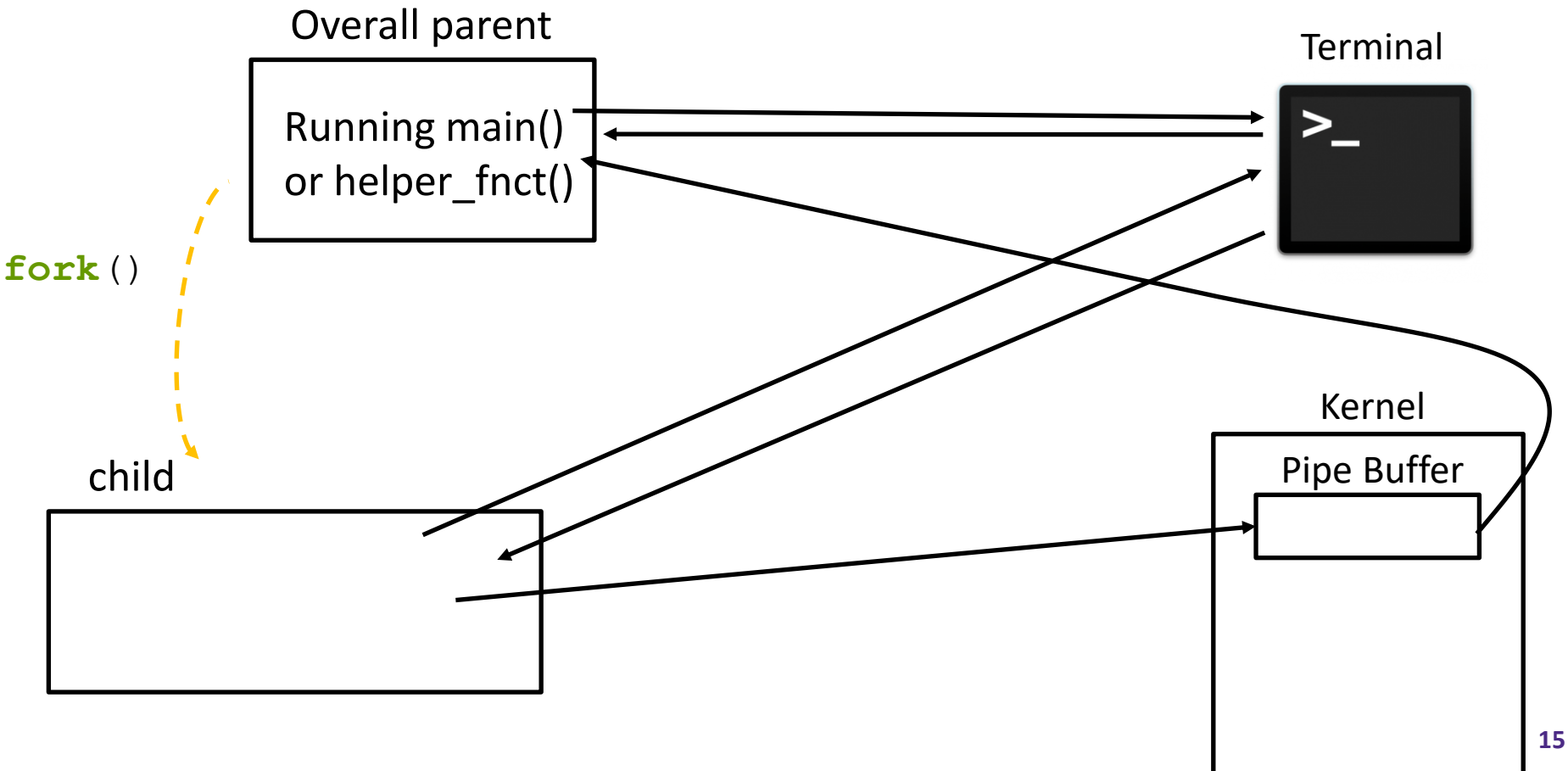
HW4 Example Line 1

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



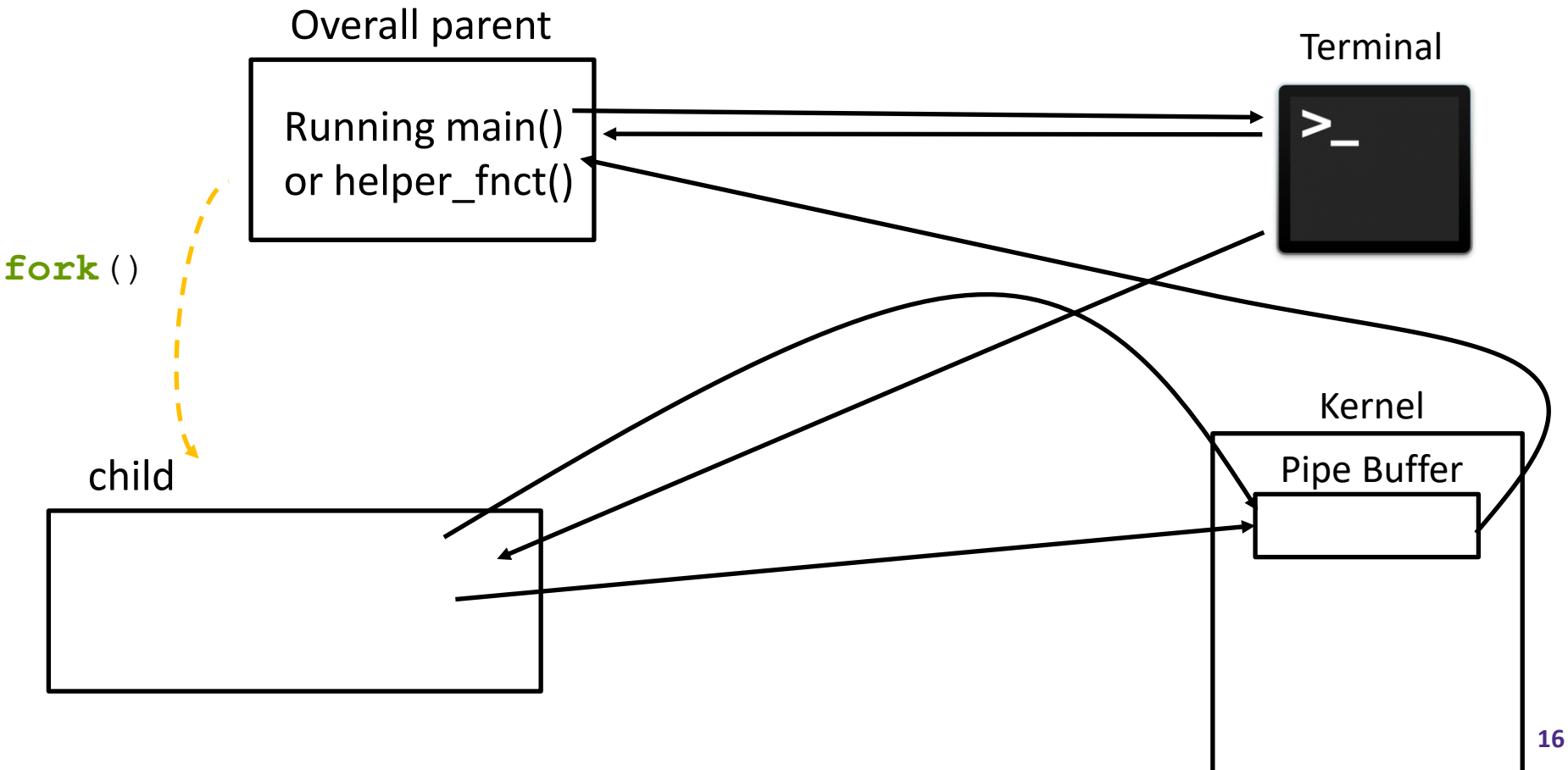
HW4 Example Line 1

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



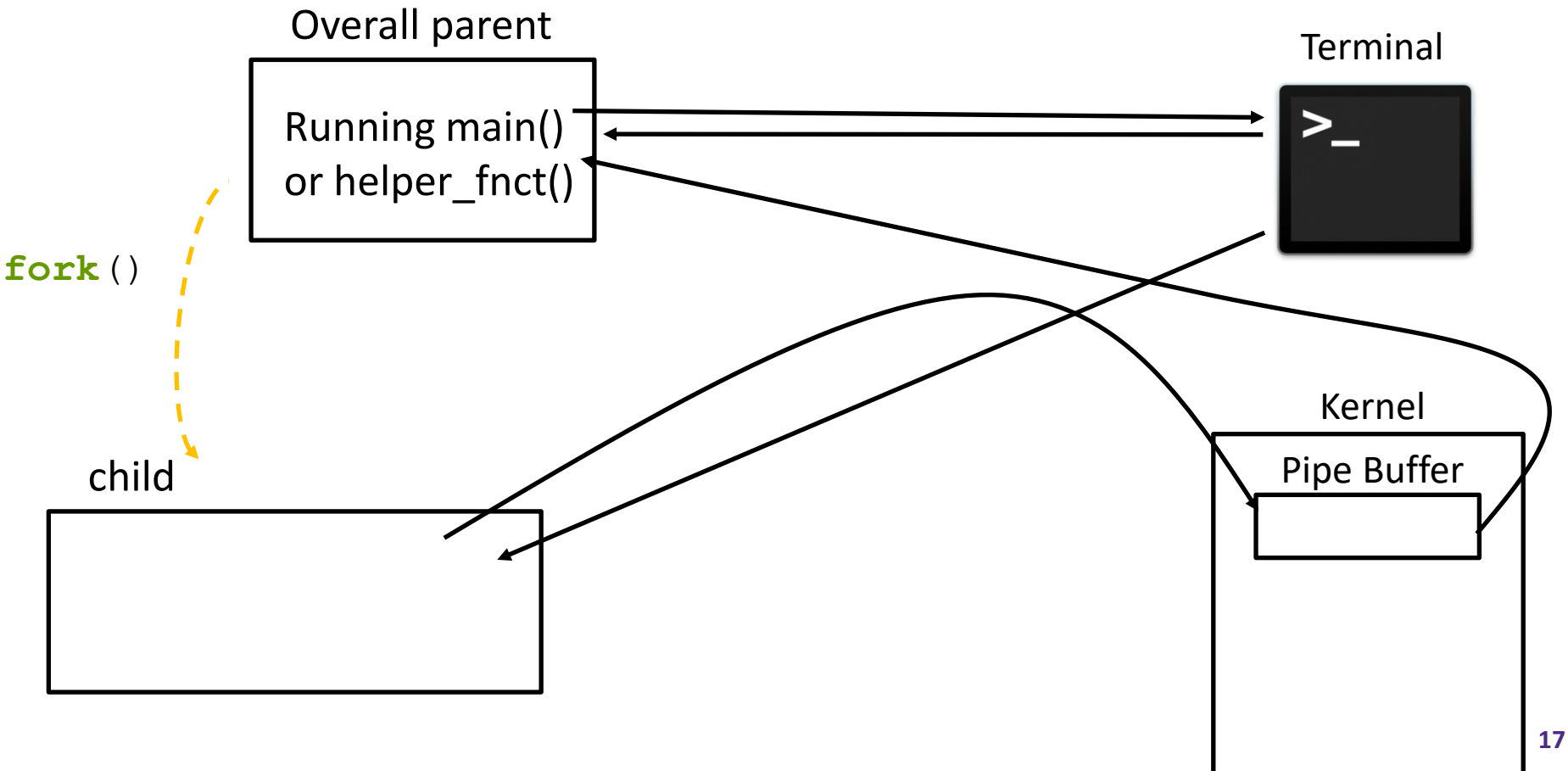
HW4 Example Line 1

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



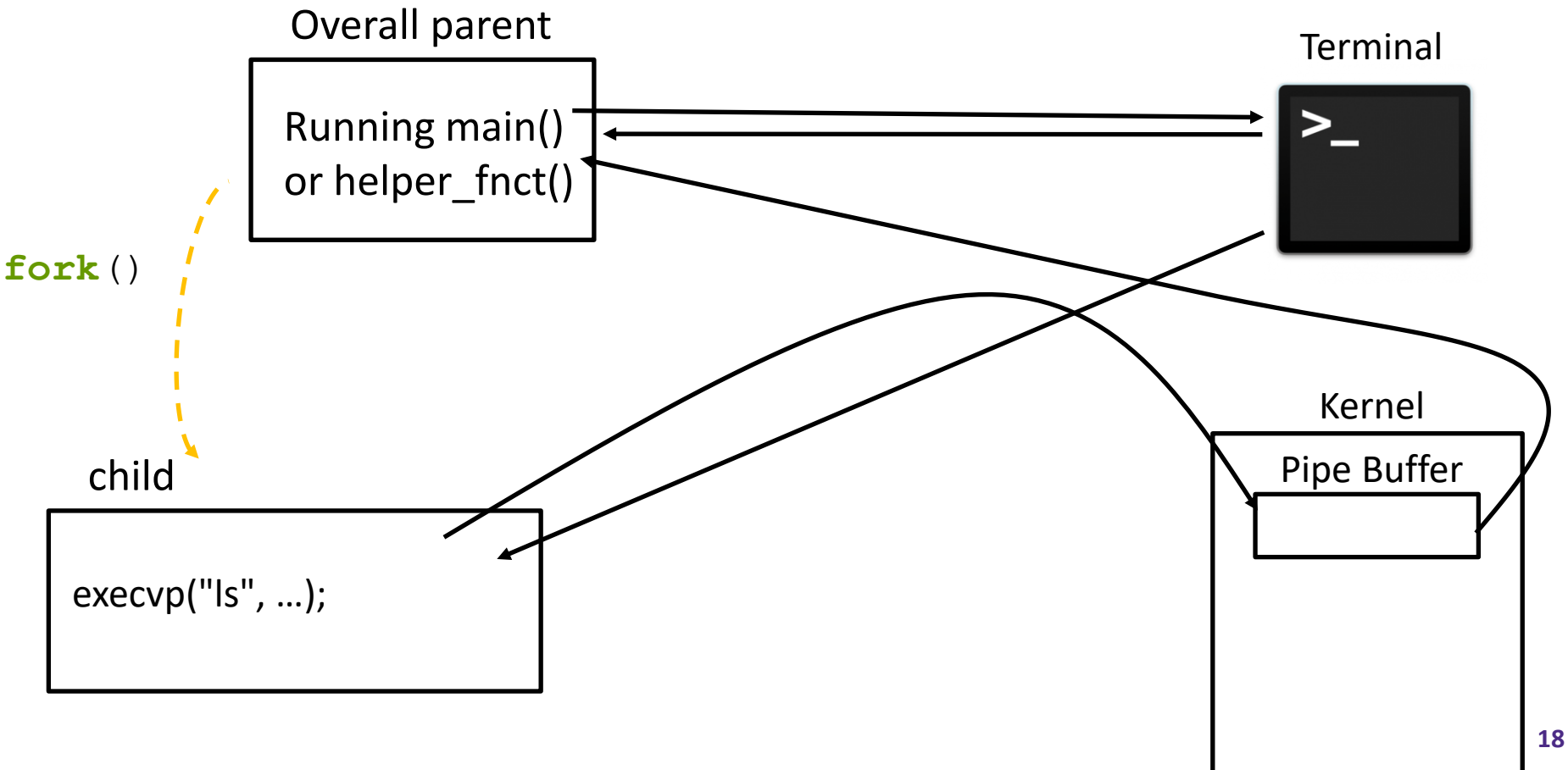
HW4 Example Line 1

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



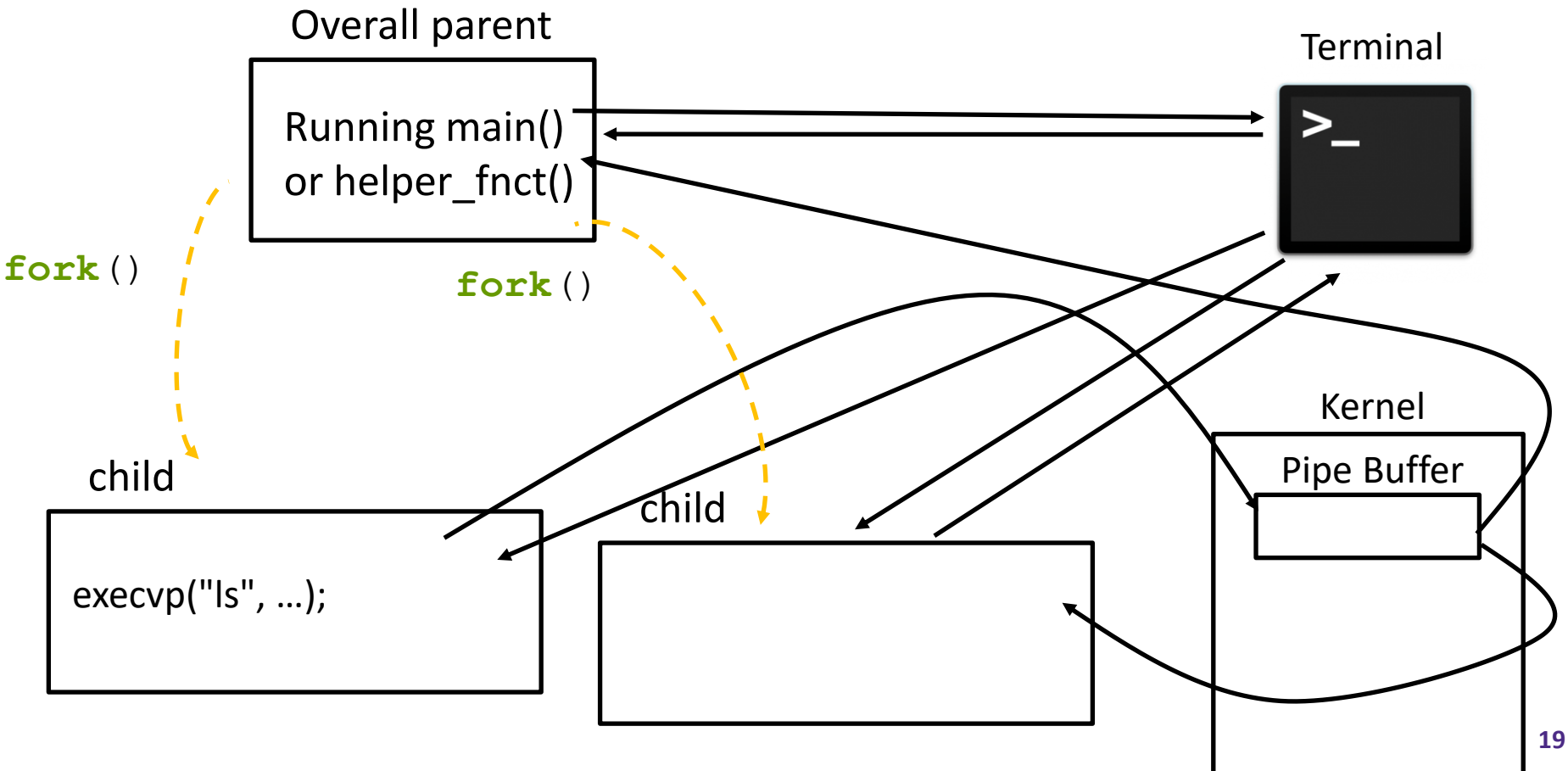
HW4 Example Line 1

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



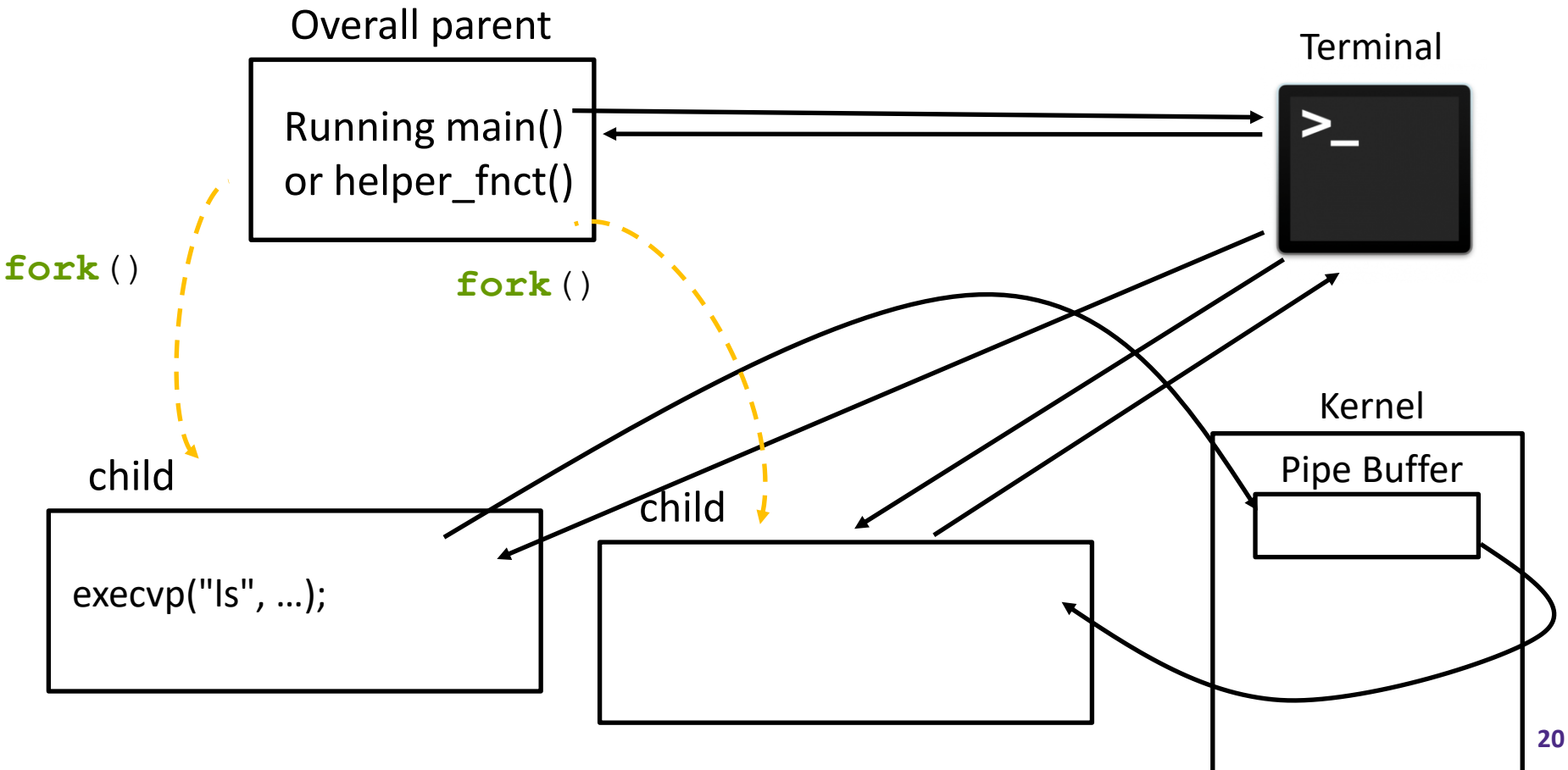
HW4 Example Line 1

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



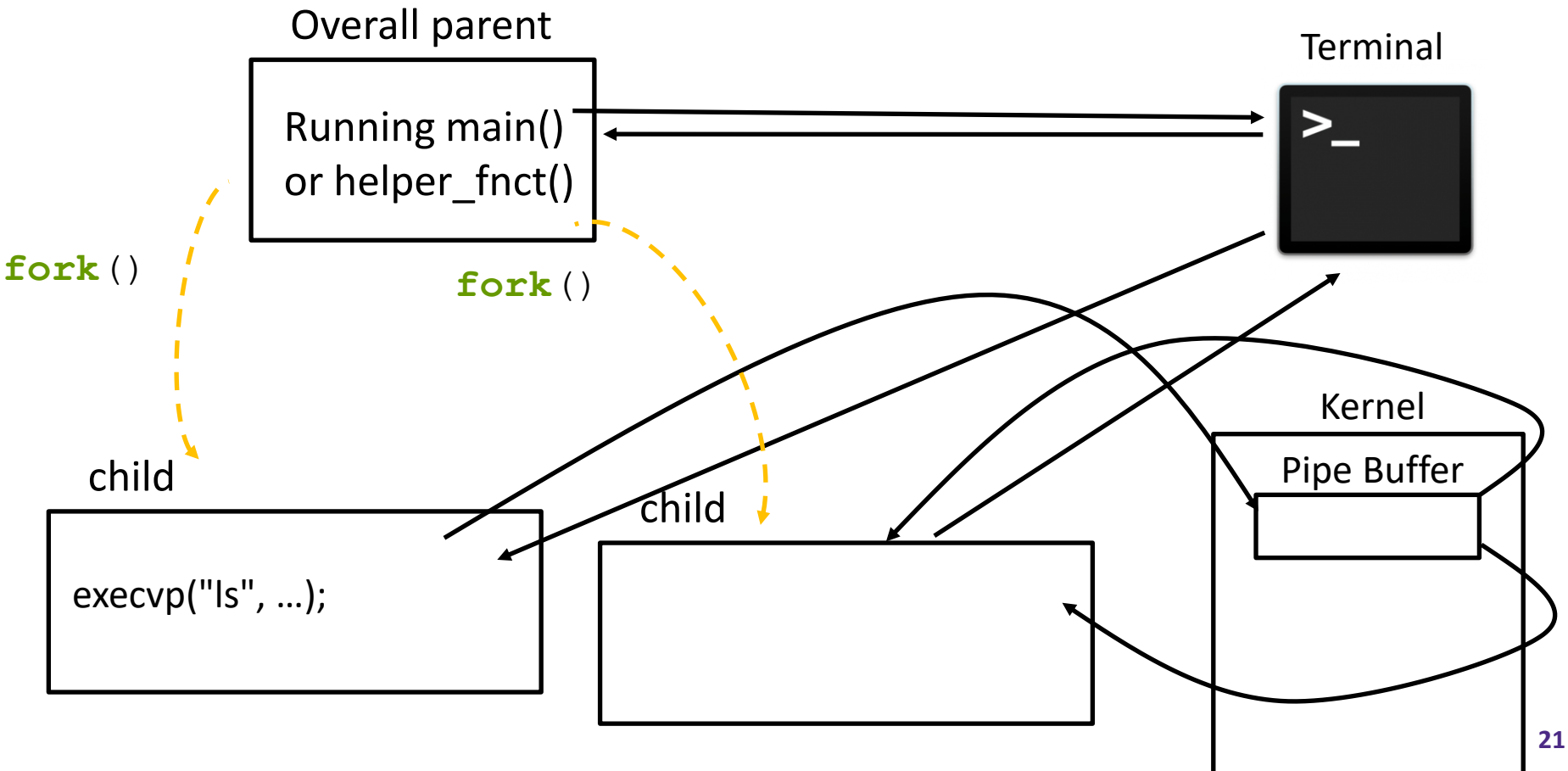
HW4 Example Line 1

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



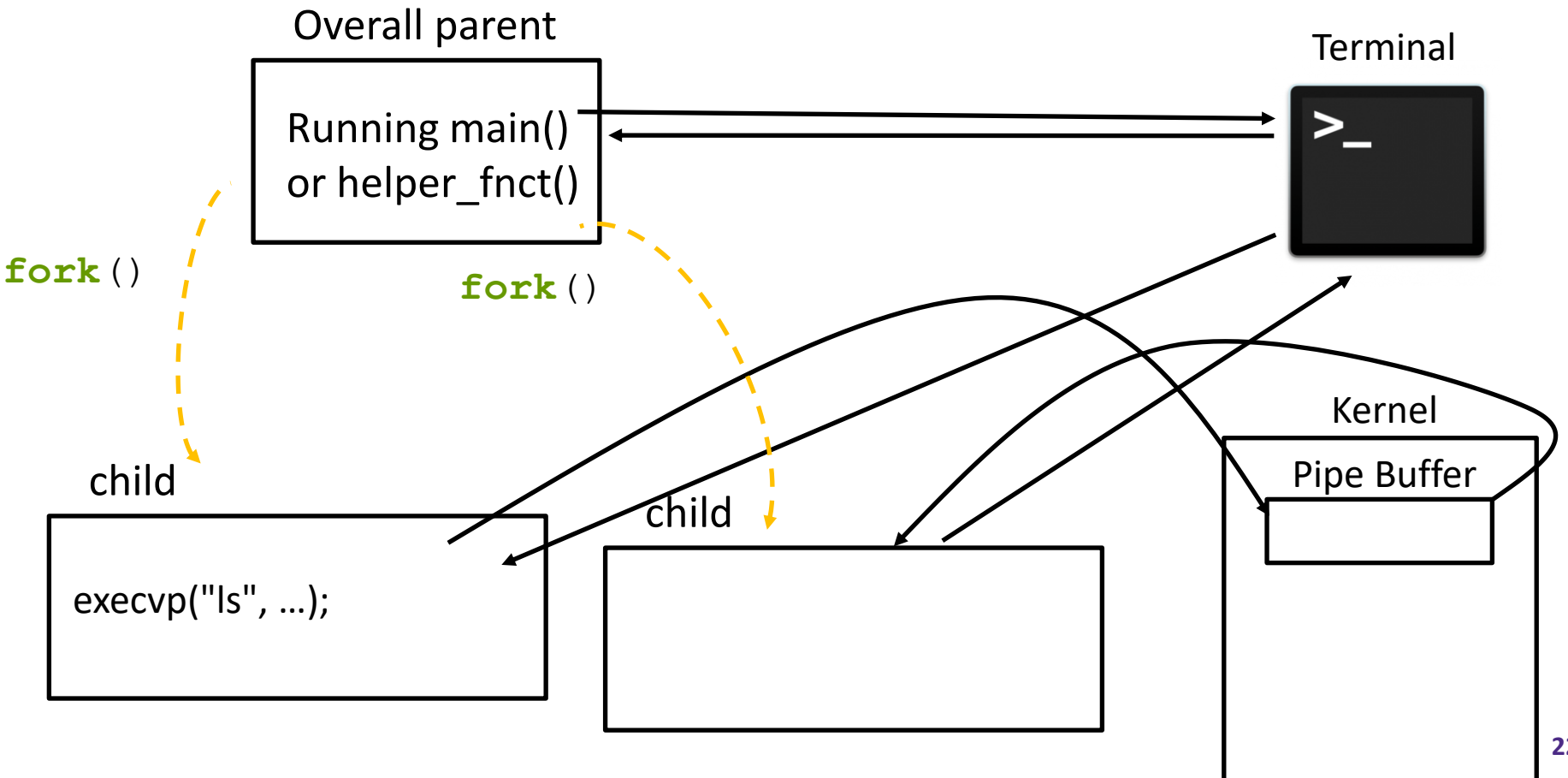
HW4 Example Line 1

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



HW4 Example Line 1

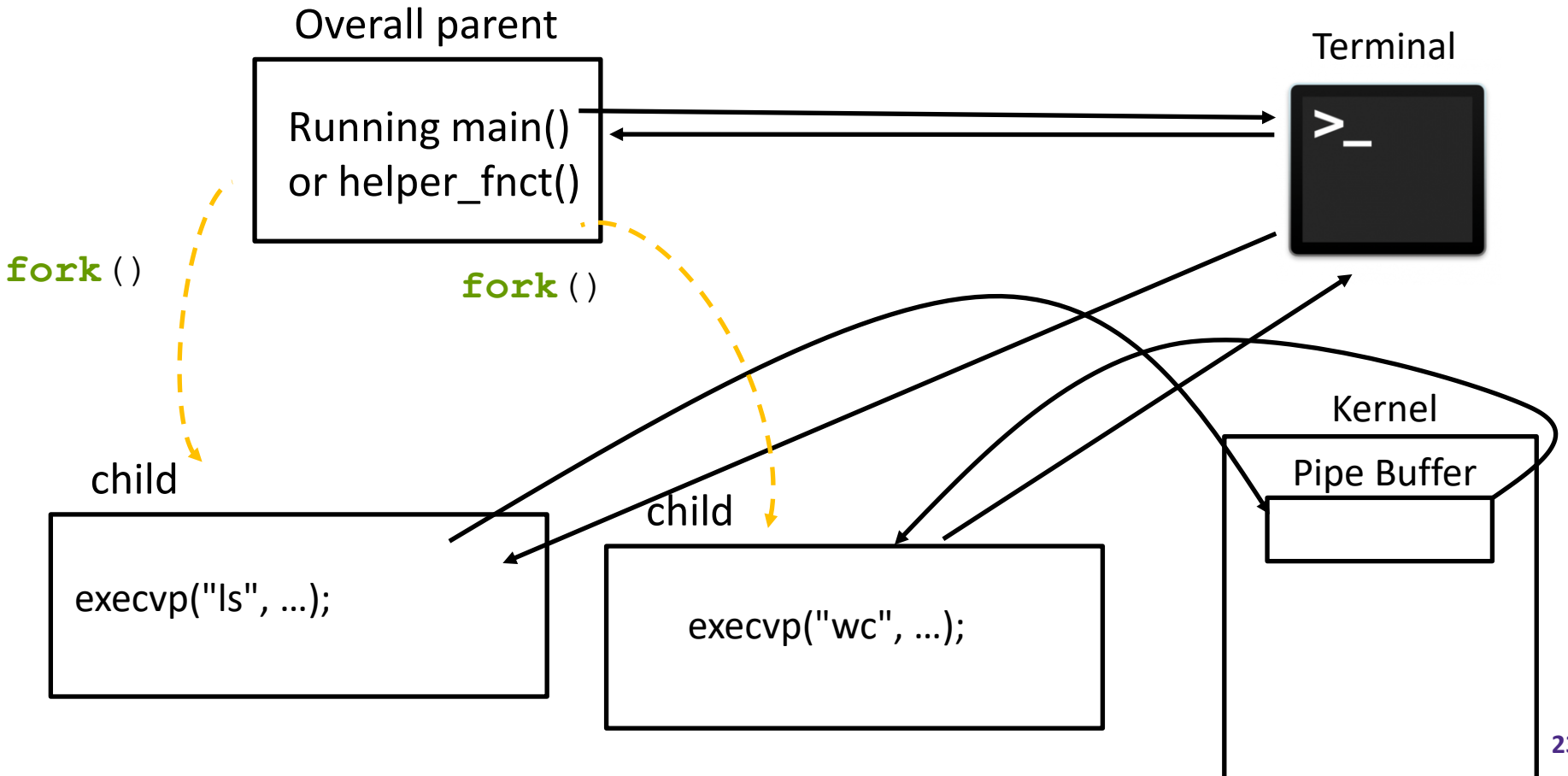
- Consider the case when a user inputs
  - `"ls | wc"`



HW4 Example Line 1

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

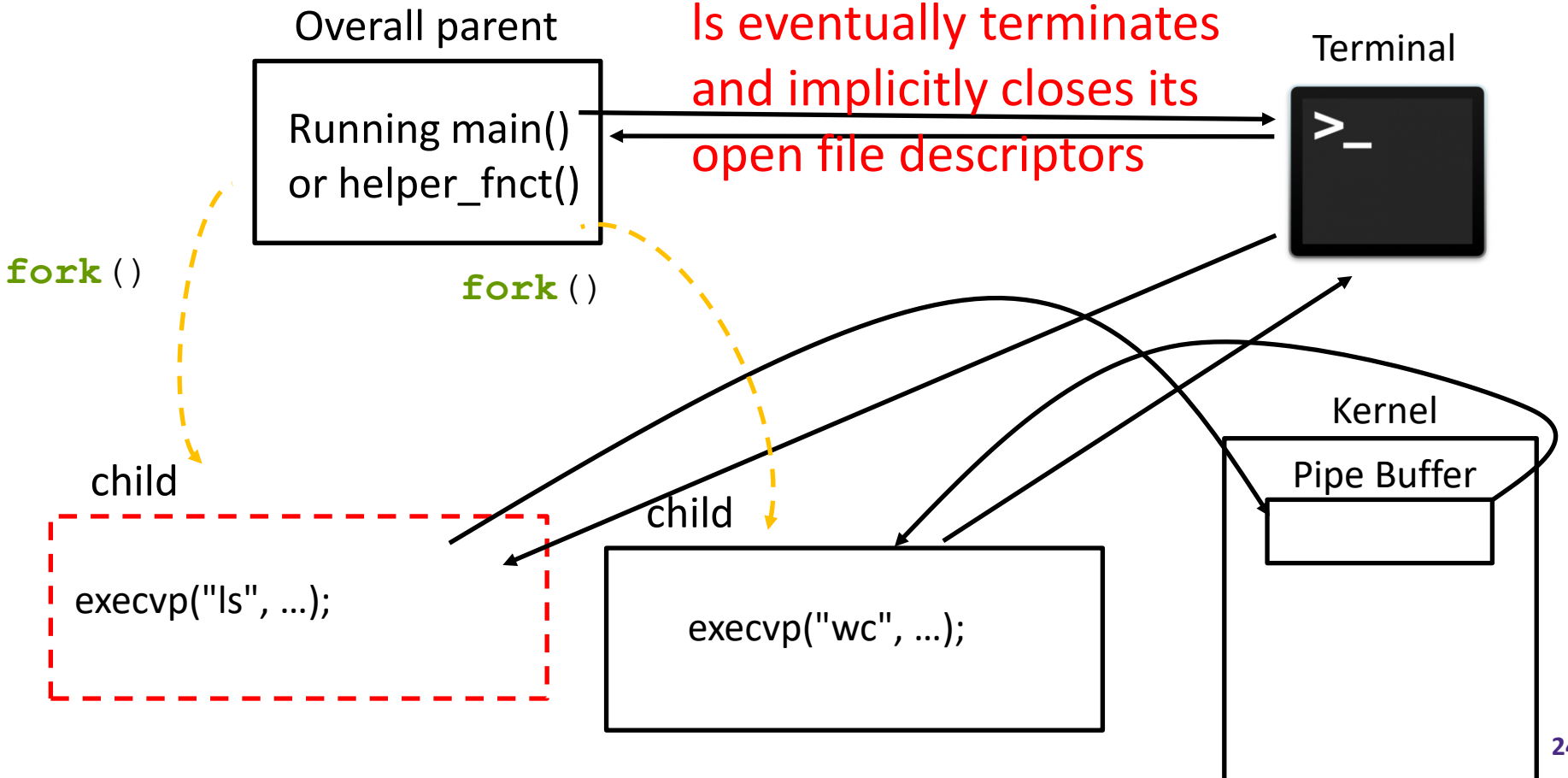
What happens when we run this code?
ls runs and wc reads what ls prints



HW4 Example Line 1

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

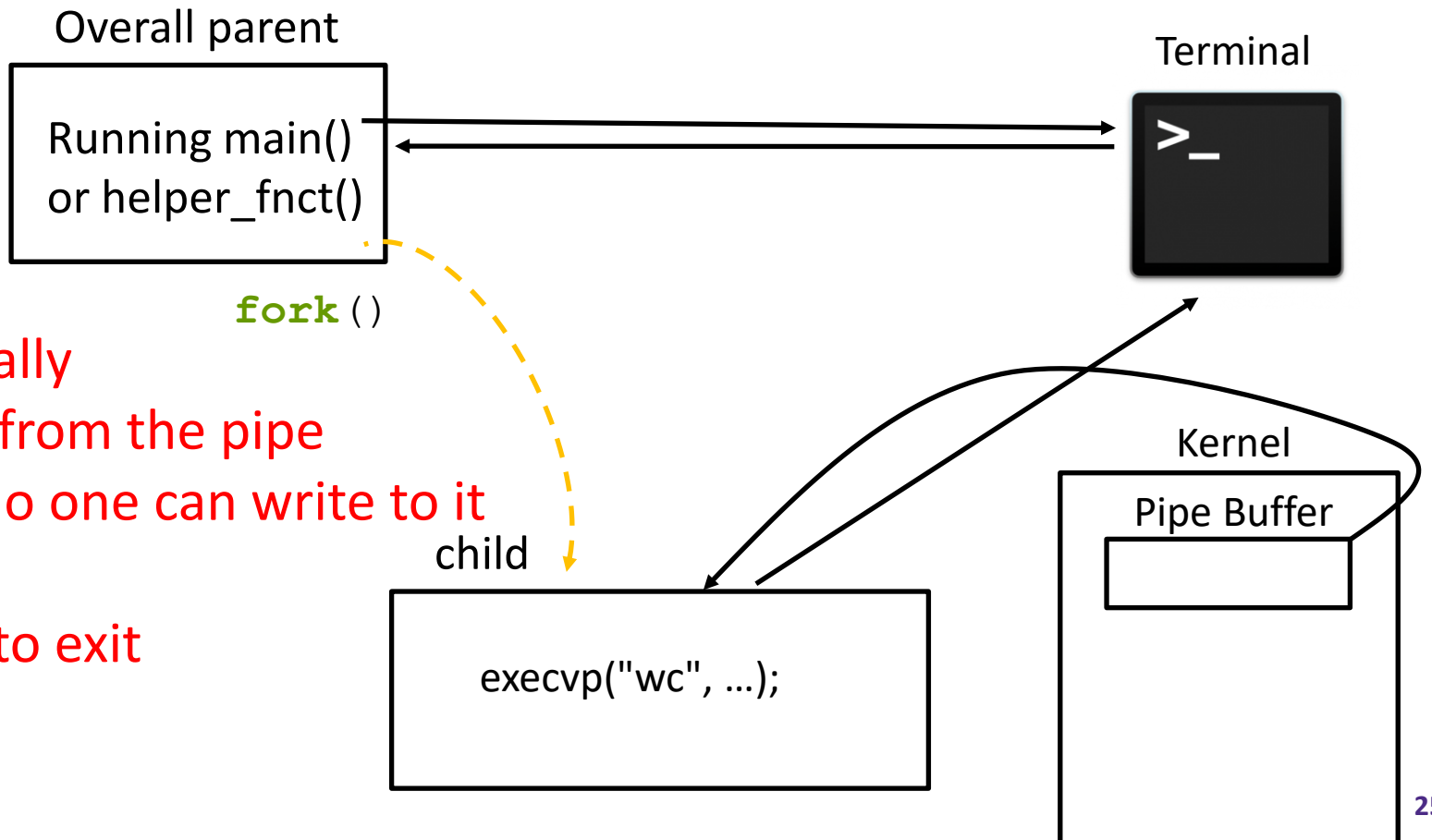
What happens when we run this code?
ls runs and wc reads what ls prints
ls eventually terminates
and implicitly closes its
open file descriptors



HW4 Example Line 1

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

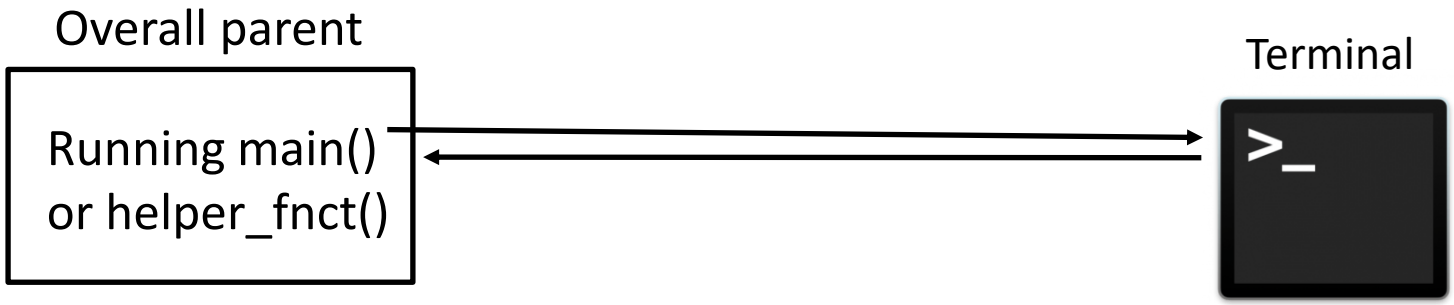
What happens when we run this code?



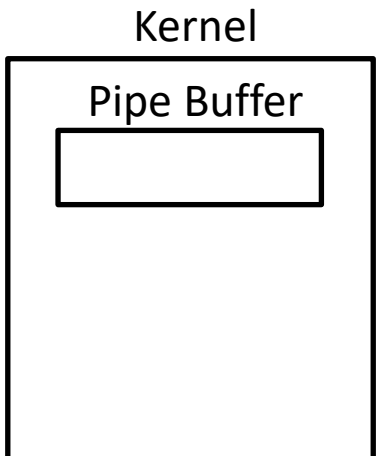
HW4 Example Line 1

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

What happens when we run this code?



pipe_shell prompts the user for the next command
After returning from waitpid on the “wc” command

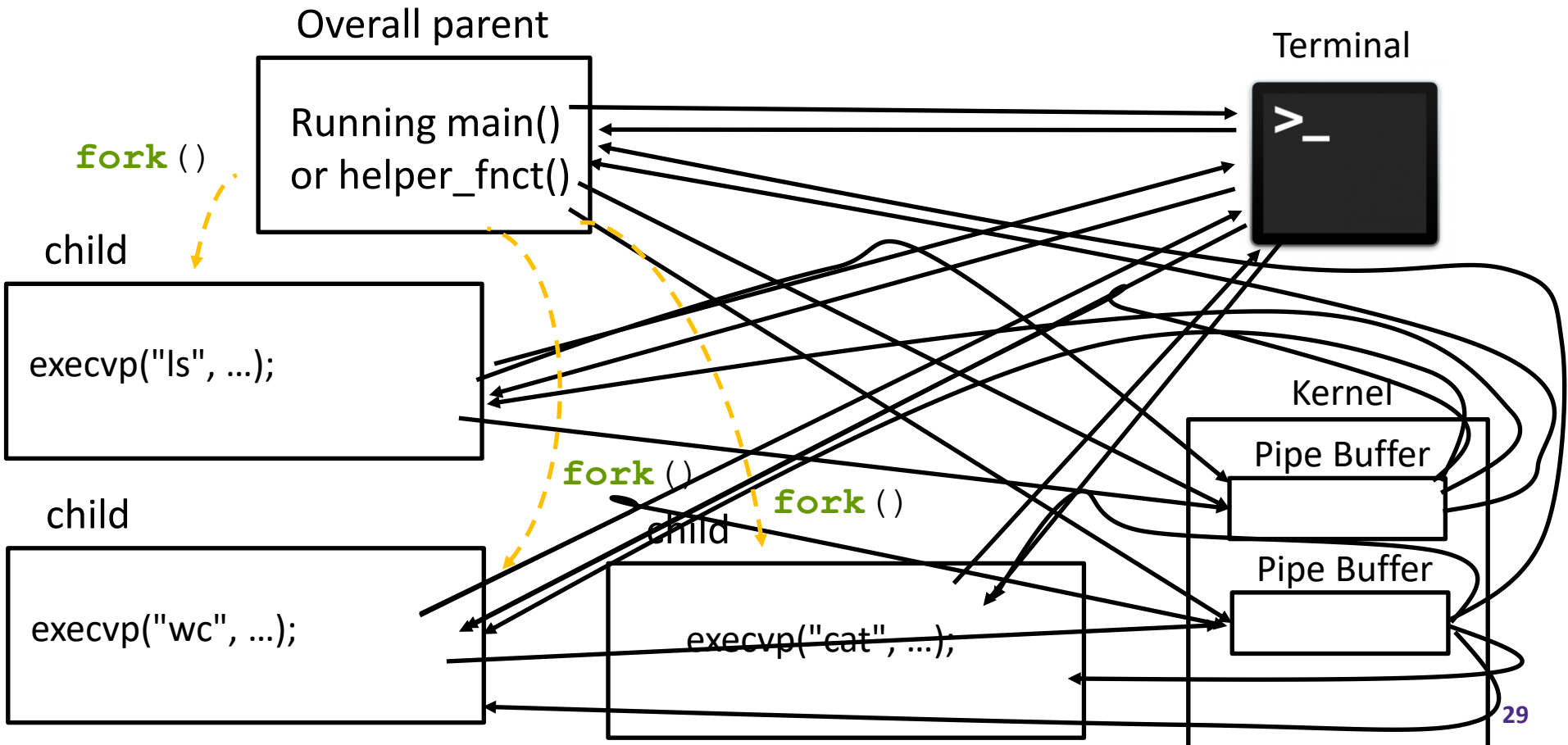


HW4 Hints

- 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

HW4 Example Line 2

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



HW4 Hints

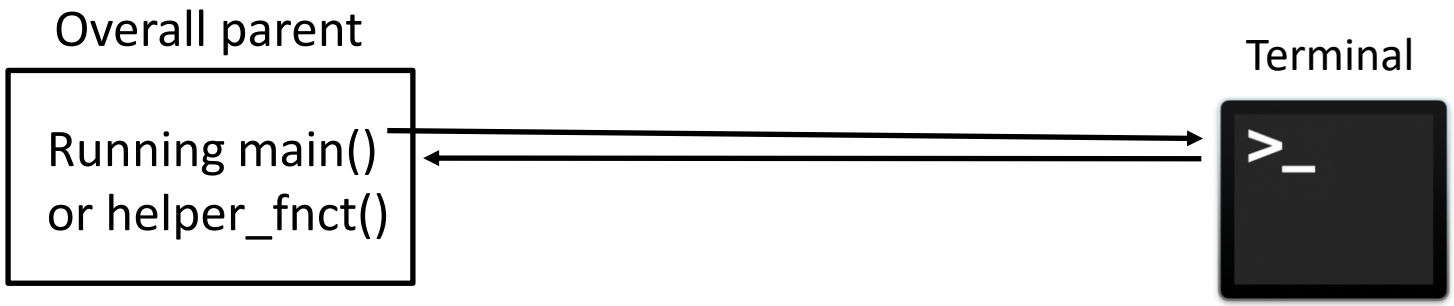
- If there are n commands in a line, there should be n-1 pipes

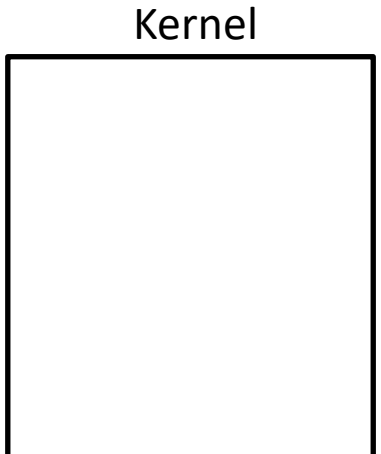
- Each pipe should be written to by **exactly** one process
- Each pipe should be read by **exactly** one process
  - Different than the one writing

- Why is this important?
  - Some programs run until they read in EOF
  - EOF can only be read from a pipe if all accesses to the write-end of the pipe are closed and there is nothing left to read.

HW4 Example Line (no pipe closing)

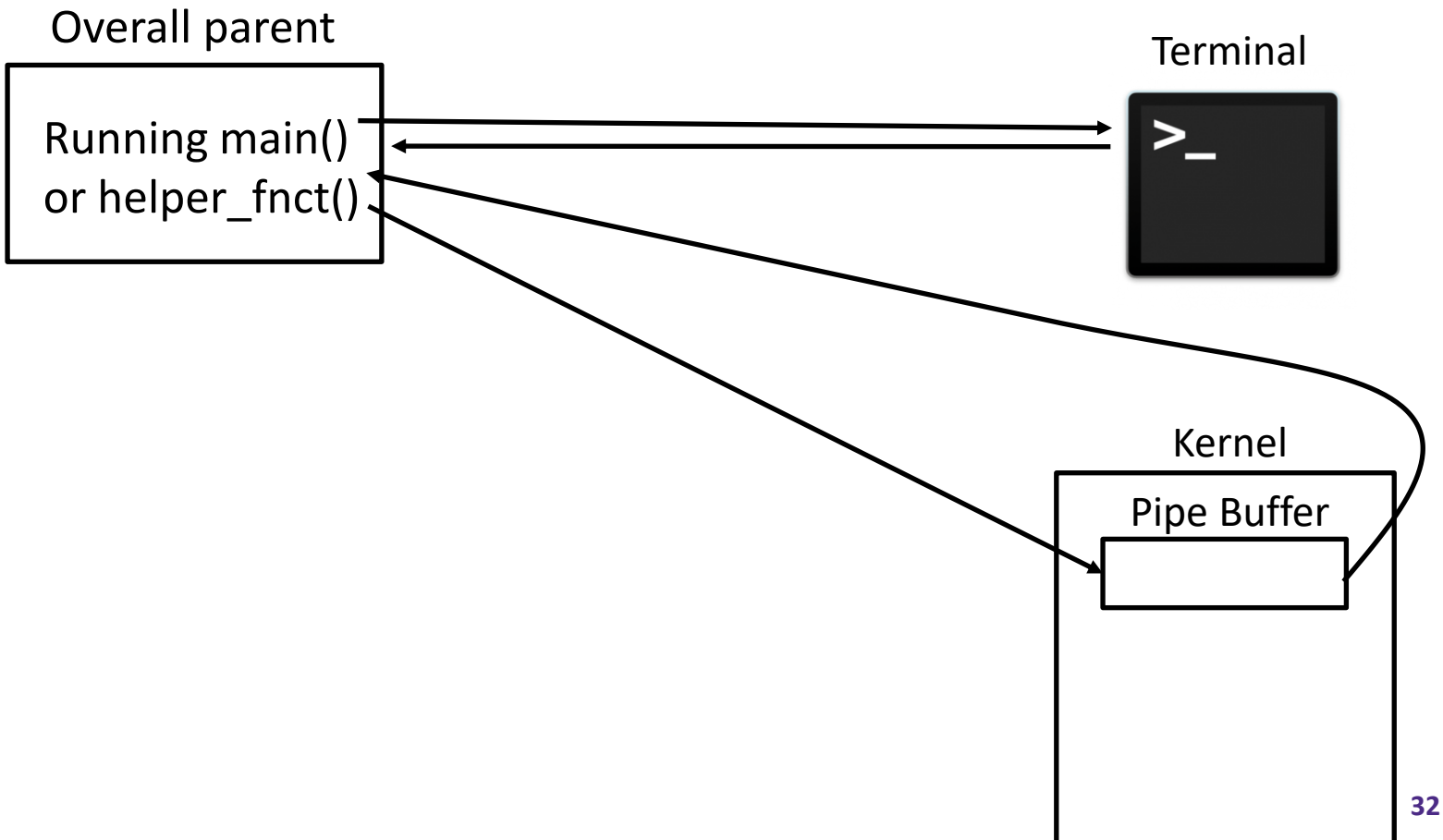
- Consider the case when a user inputs
  - `"ls | cat"`





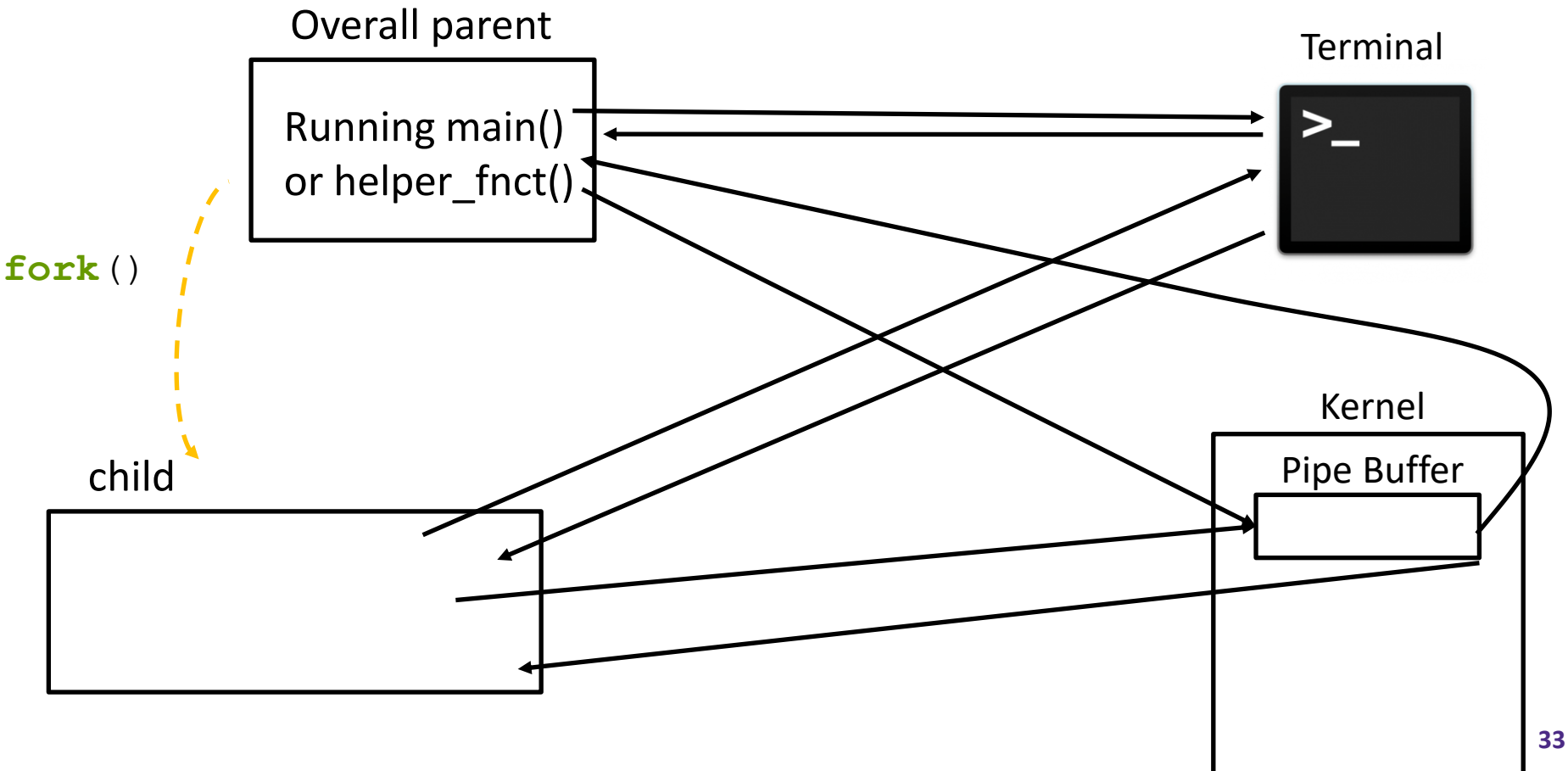
HW4 Example Line (no pipe closing)

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



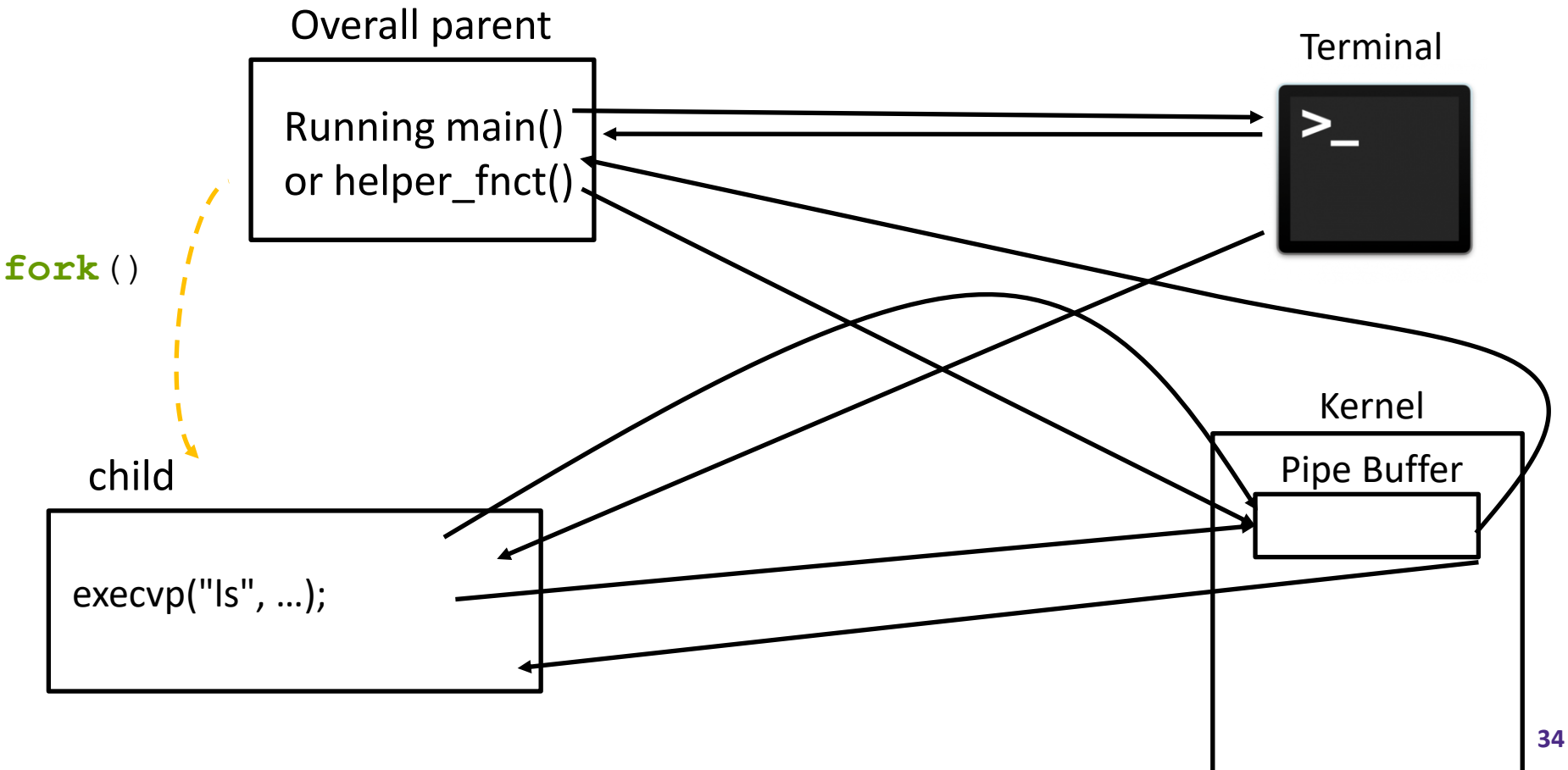
HW4 Example Line (no pipe closing)

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



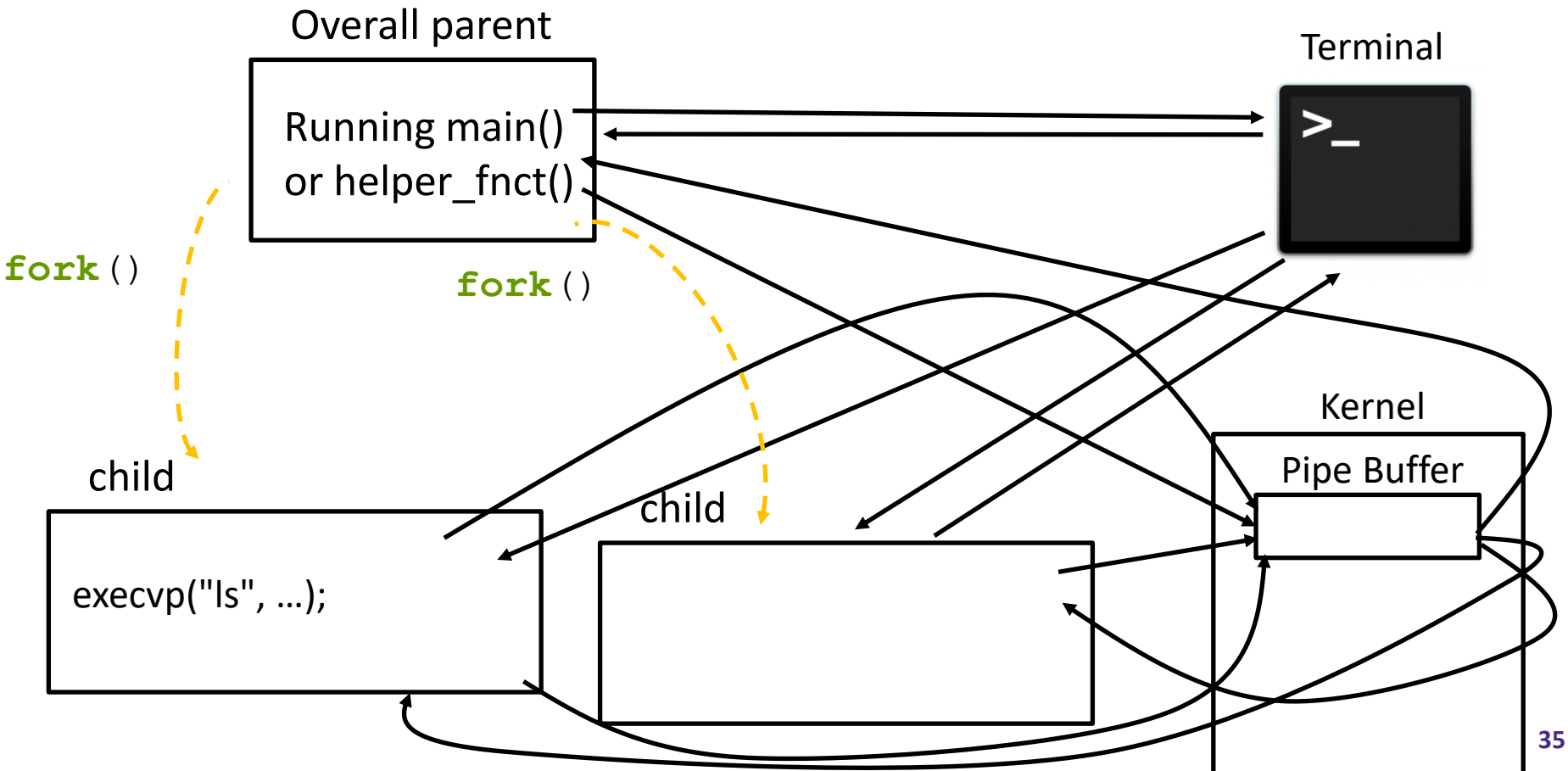
HW4 Example Line (no pipe closing)

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



HW4 Example Line (no pipe closing)

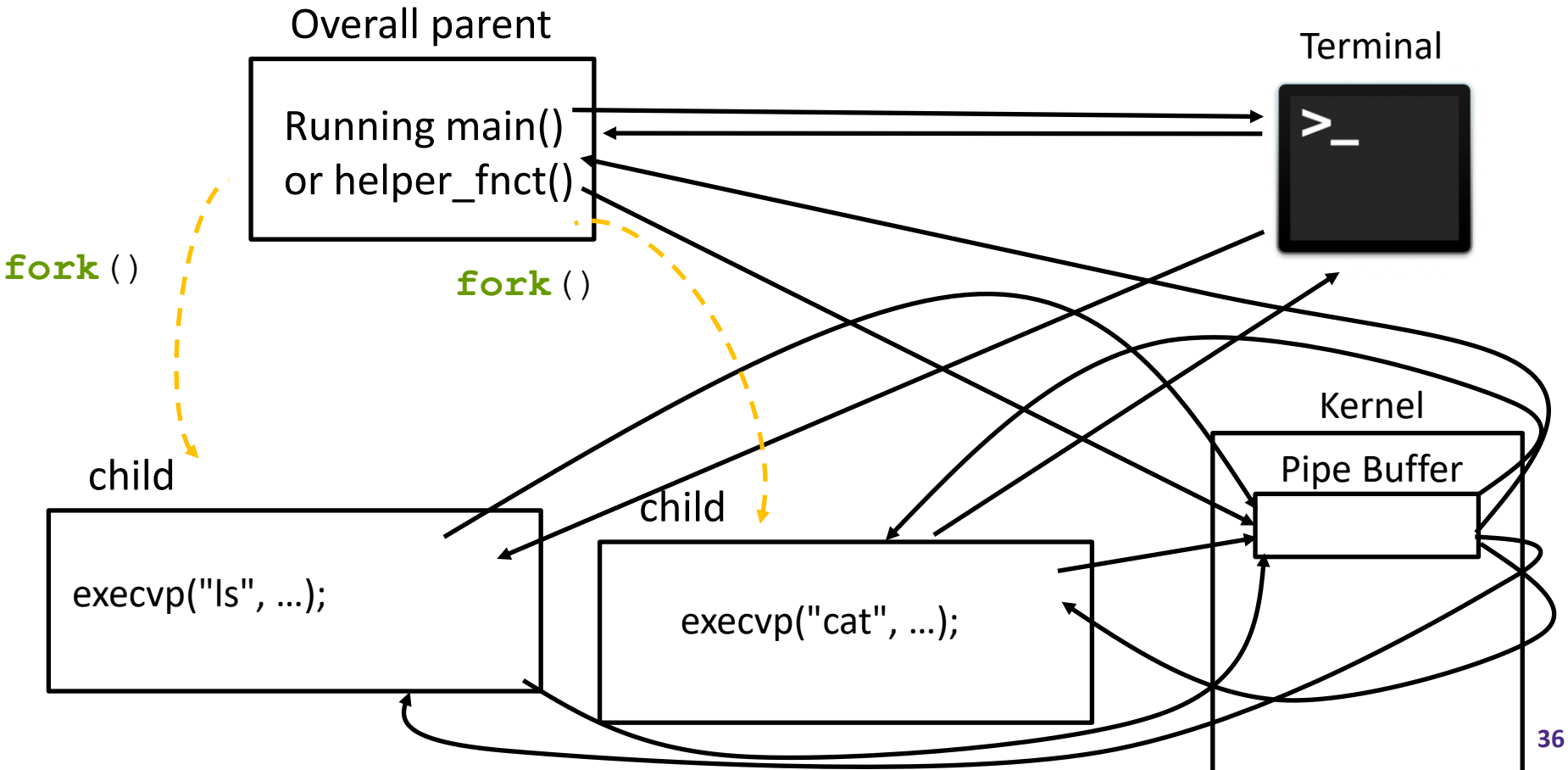
- Consider the case when a user inputs
  - `"ls | cat"`



HW4 Example Line (no pipe closing)

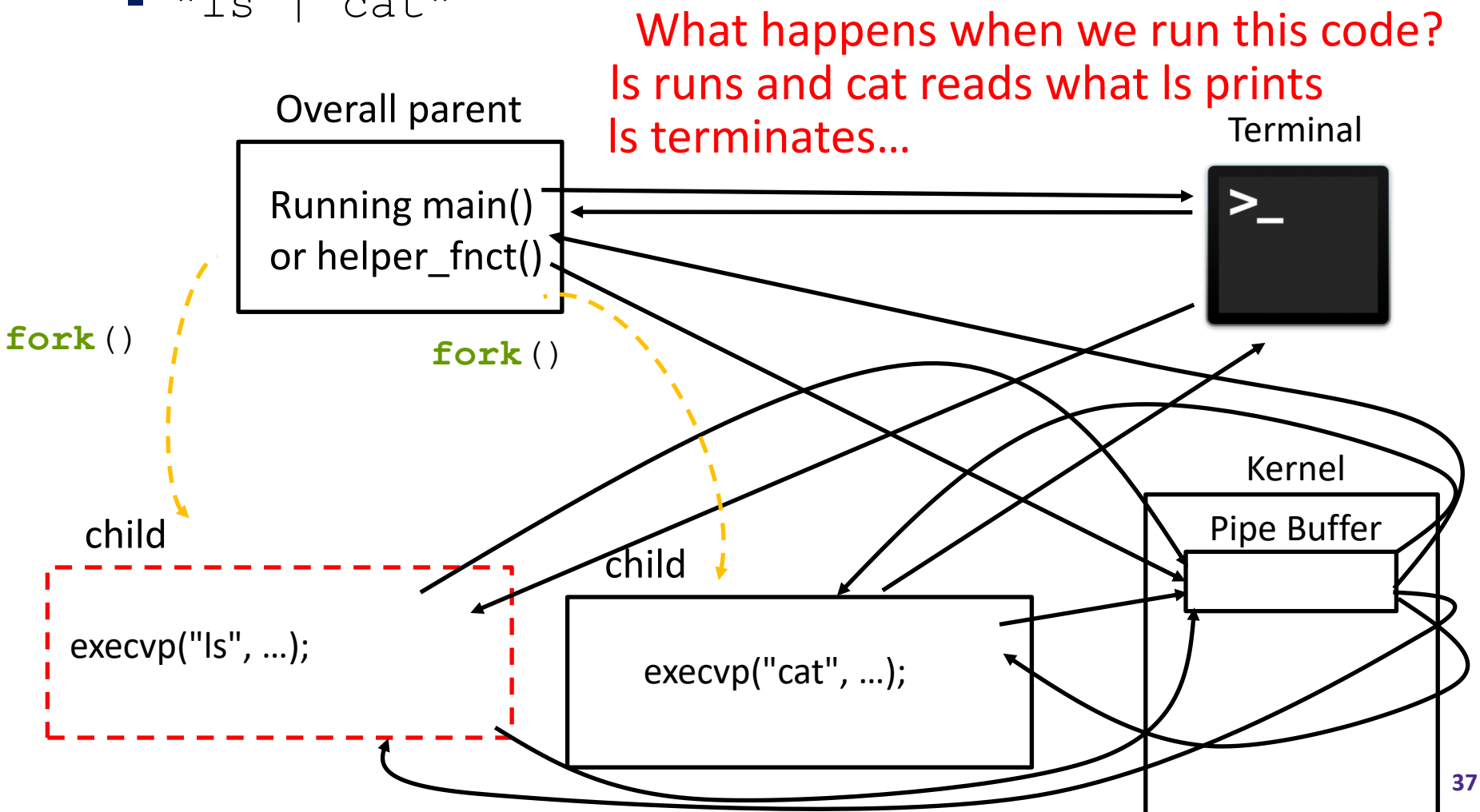
- Consider the case when a user inputs
  - `"ls | cat"`

What happens when we run this code?



HW4 Example Line (no pipe closing)

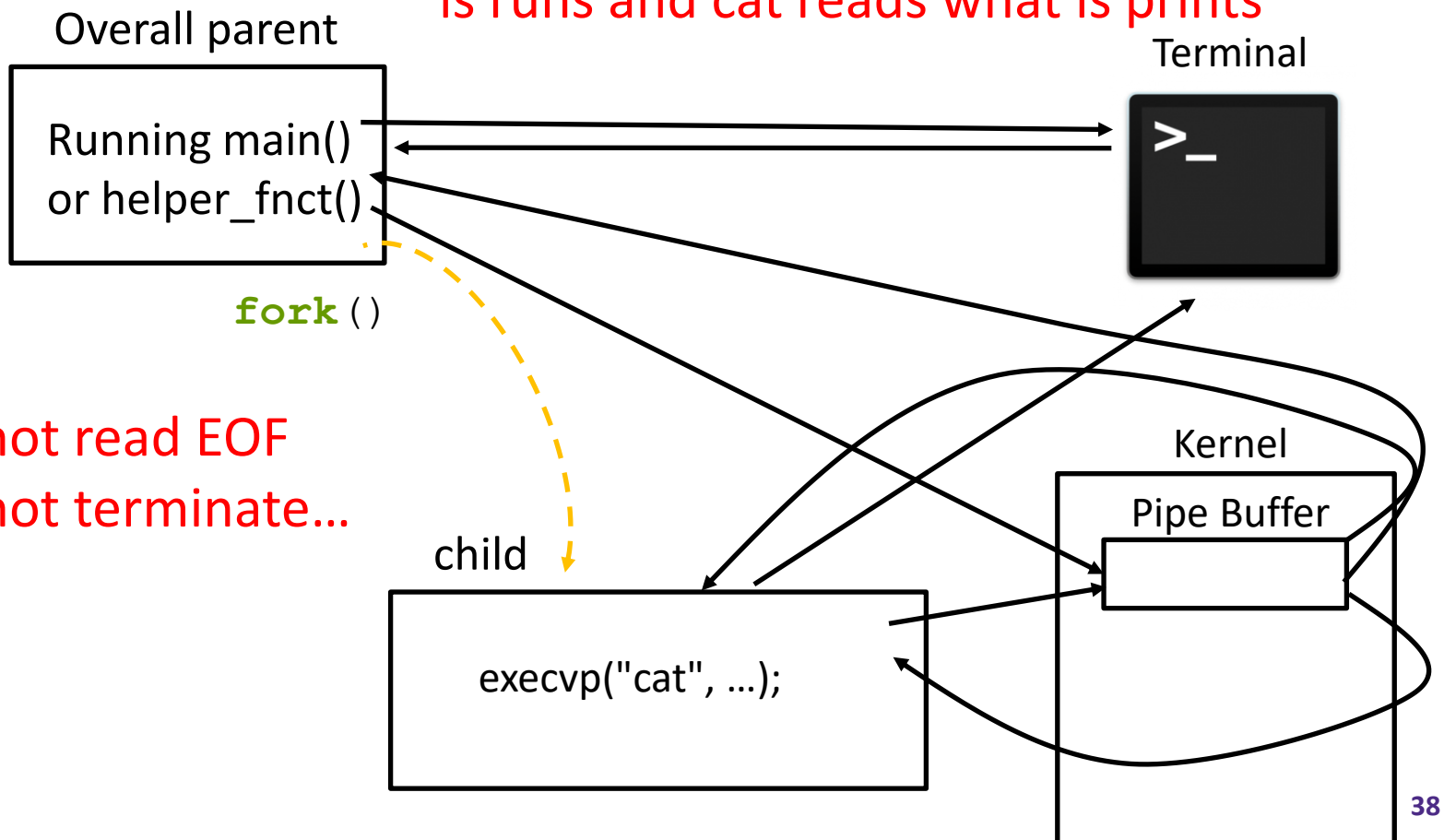
- Consider the case when a user inputs
  - `"ls | cat"`



HW4 Example Line (no pipe closing)

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

What happens when we run this code?
ls runs and cat reads what ls prints



Lecture Outline

- More HW4
- **Polymorphism (start)**
  - **Inheritance motivation & C++ Syntax**
  - Polymorphism & Dynamic Dispatch

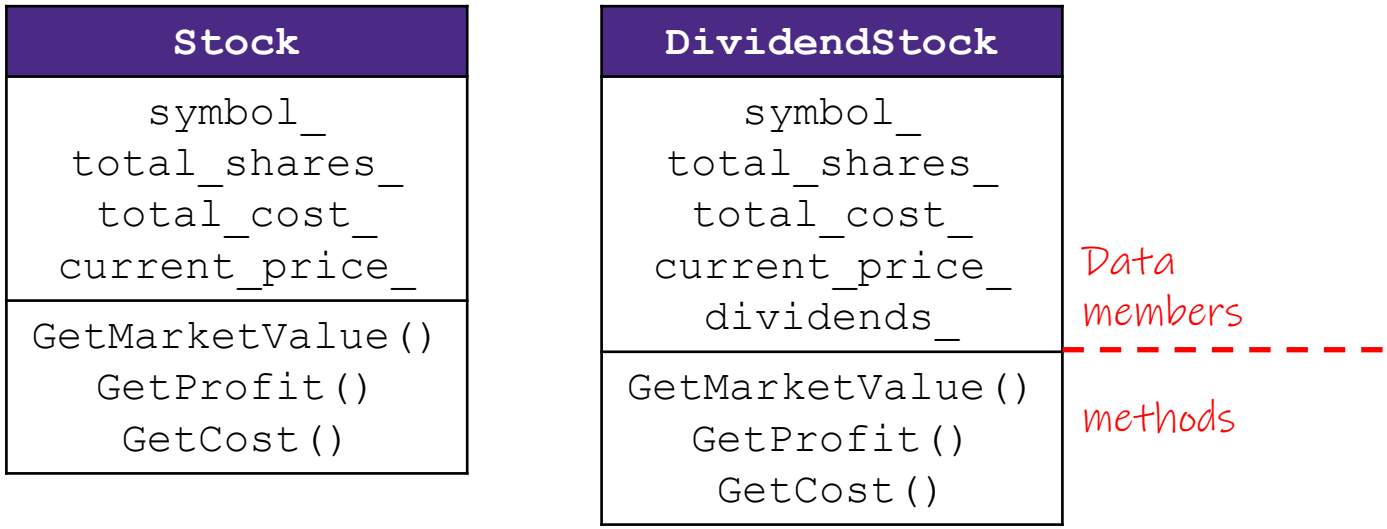
Stock Portfolio Example

- A portfolio represents a person's financial investments
  - Each *asset* has a cost (*i.e.* how much was paid for it) and a market value (*i.e.* how much it is worth)
    - The difference between the cost and market value is the *profit* (or loss)
  - Different assets compute market value in different ways
    - A **stock** that you own has a ticker symbol (*e.g.* "GOOG"), a number of shares, share price paid, and current share price
    - A **dividend stock** is a stock that also has dividend payments
    - **Cash** is an asset that never incurs a profit or loss

(Credit: thanks to Marty Stepp for this example)

Design Without Inheritance

- One class per asset type:



  - Redundant!
  - Cannot treat multiple investments together
    - *e.g.* can't have an array or `vector` of different assets

- See sample code in `initial.tar`

Inheritance

- A parent-child "is-a" relationship between classes
  - A child (derived class) extends a parent (base class)

- Terminology:

Java	C++
Superclass	Base Class
Subclass	Derived Class

Subclass inherits from super class. (Superclass is "higher" in the hierarchy)

Derived class inherits from base class. (base class is "higher" in the hierarchy)

  - Mean the same things. You'll hear both.

Inheritance

- A parent-child "is-a" relationship between classes
  - A child (derived class) extends a parent (base class)

- Benefits:
  - Code reuse
    - Children can automatically inherit code from parents
  - Polymorphism
    - Ability to redefine existing behavior but preserve the interface
    - Children can override the behavior of the parent
    - Others can make calls on objects without knowing which part of the inheritance tree it is in
  - Extensibility
    - Children can add behavior

Like Java: Access Modifiers

- `public`: visible to all other classes
- `protected`: visible to current class and its *derived* classes
- `private`: visible only to the current class

- Use `protected` for class members only when
  - Class is designed to be extended by derived classes
  - Derived classes must have access but clients should not be allowed

Class Derivation List

- Comma-separated list of classes to inherit from:

```
#include "BaseClass.h"

class Name : public BaseClass {
  ...
};
```

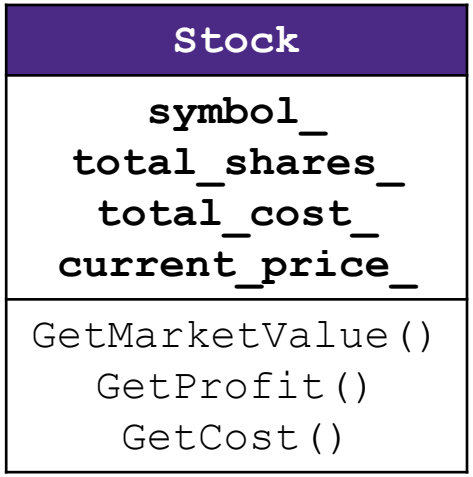
  - Focus on single inheritance, but *multiple inheritance* possible

: public Base1, public Base2 {

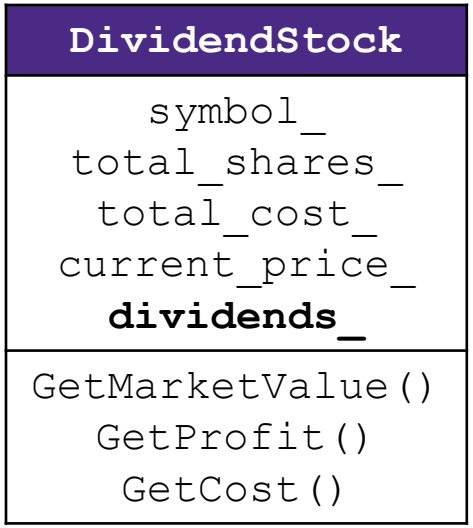
- Almost always you will want public inheritance
  - Acts like `extends` does in Java
  - Any member that is non-private in the base class is the same in the derived class; both *interface and implementation inheritance*

Except that constructors, destructors, copy constructor, and assignment operator are *never* inherited

Back to Stocks

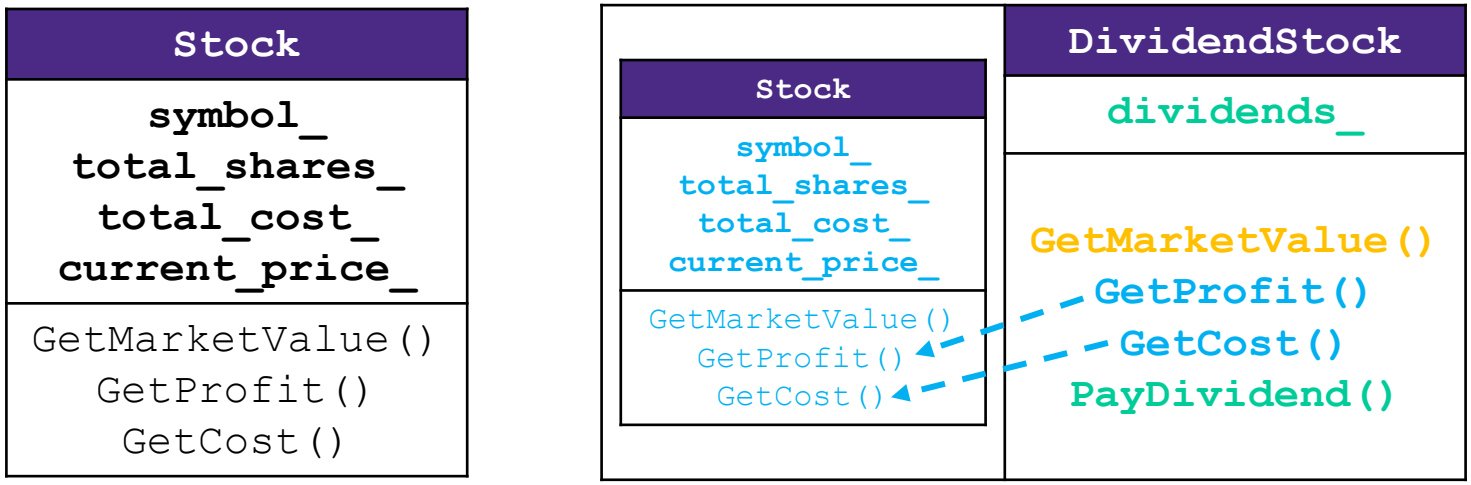


BASE



DERIVED

Back to Stocks



- A derived class:
  - **Inherits** the behavior and state (specification) of the base class
  - **Overrides** some of the base class' member functions (opt.)
  - **Extends** the base class with new member functions, variables (opt.)

Lecture Outline

- More HW4
- **Polymorphism (start)**
  - Inheritance motivation & C++ Syntax
  - **Polymorphism & Dynamic Dispatch**

Polymorphism in C++

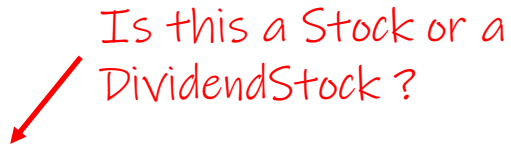
- In Java: `PromisedType var = new ActualType();`
  - `var` is a reference (different term than C++ reference) to an object of `ActualType` on the Heap
  - `ActualType` must be the same class or a subclass of `PromisedType`
- In C++: `PromisedType* var_p = new ActualType();`
  - `var_p` is a *pointer* to an object of `ActualType` on the Heap
  - `ActualType` must be the same or a derived class of `PromisedType`
  - (also works with references)
  - `PromisedType` defines the *interface* (*i.e.* what can be called on `var_p`), but `ActualType` may determine which *version* gets invoked

Dynamic Dispatch (like Java)

- Usually, when a derived function is available for an object, we want the derived function to be invoked
  - This requires a *run time* decision of what code to invoke

- A member function invoked on an object should be the *most-derived function* accessible to the object's visible type
  - Can determine what to invoke from the *object* itself

- Example:
  - `void PrintStock(Stock* s) { s->Print(); }`
  - Calls the appropriate `Print()` without knowing the actual type of `*s`, other than it is some sort of `Stock`



Requesting Dynamic Dispatch (C++)

- Prefix the member function declaration with the `virtual` keyword
  - Derived/child functions don't need to repeat `virtual`, but was traditionally good style to do so
  - This is how method calls work in Java (no virtual keyword needed)
  - You almost always want functions to be virtual
- `override` keyword (C++11)
  - Tells compiler this method should be overriding an inherited virtual function – *always* use if available
  - Prevents overloading vs. overriding bugs
- Both of these are technically *optional* in derived classes
  - Be consistent and follow local conventions (Google Style Guide says no `virtual` if `override`)

Dynamic Dispatch Example

- When a member function is invoked on an object:
  - The *most-derived function* accessible to the object's visible type is invoked (decided at run time based on actual type of the object)

```
double DividendStock::GetMarketValue() const {
  return get_shares() * get_share_price() + dividends_;
}

double "DividendStock"::GetProfit() const {  // inherited
  return GetMarketValue() - GetCost();
}   Should call DividendStock::GetMarketValue()
```

DividendStock.cc

Inherited from stock

```
double Stock::GetMarketValue() const {
  return get_shares() * get_share_price();
}

double Stock::GetProfit() const {
  return GetMarketValue() - GetCost();
}
```

Stock.cc

Dynamic Dispatch Example

```
#include "Stock.h"
#include "DividendStock.h"

DividendStock dividend();
DividendStock* ds = &dividend;
Stock* s = &dividend;    // why is this allowed?

// Invokes DividendStock::GetMarketValue()
ds->GetMarketValue();

// Invokes DividendStock::GetMarketValue()
s->GetMarketValue();

// invokes Stock::GetProfit(), since that method is inherited.
// Stock::GetProfit() invokes DividendStock::GetMarketValue(),
// since that is the most-derived accessible function.
s->GetProfit();
```

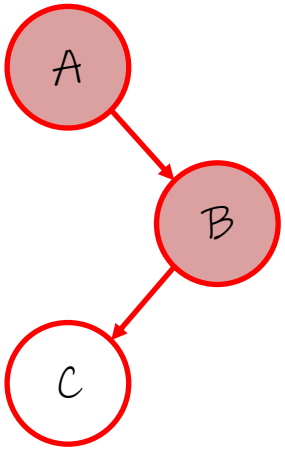
A DividendStock "is-a" Stock, and has every part of Stock's interface

Most-Derived

```
class A {
 public:
  // Foo will use dynamic dispatch
  virtual void Foo();
};

class B : public A {
 public:
  // B::Foo overrides A::Foo
  virtual void Foo();
};

class C : public B {
  // C inherits B::Foo()
};
```

```
void Bar() {
  A* a_ptr;
  C c;

  a_ptr = &c;

  // Whose Foo() is called?
  a_ptr->Foo(); // B::Foo
}
```

Has Foo definition



pollev.com/tqm

- Whose `Foo()` is called?

	Q1	Q2
A.	A	B
B.	A	D
C.	B	B
D.	B	D
E.	We're lost…	

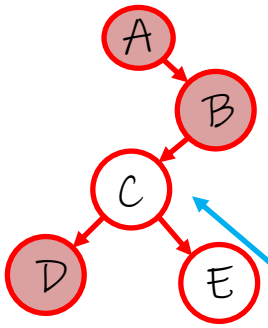
```
void Bar() {
  A* a_ptr;
  C c;
  E e;

  // Q1:
  a_ptr = &c;
  a_ptr->Foo();

  // Q2:
  a_ptr = &e;
  a_ptr->Foo();
}
```

```
class A {
 public:
  virtual void Foo();
};

class B : public A {
 public:
  virtual void Foo();
};

class C : public B {
};

class D : public C {
 public:
  virtual void Foo();
};

class E : public C {
};
```

Poll Everywhere

pollev.com/tqm

- Whose **Foo ( )** is called?



	Q1	Q2
A.	A	B
B.	A	D
C.	B	B
D.	B	D
E.	We're lost…	

```
void Bar() {
  A* a_ptr;
  C c;
  E e;

  // Q1:
  a_ptr = &c;
  a_ptr->Foo();
          B::Foo()
  // Q2:
  a_ptr = &e;
  a_ptr->Foo();
}         B::Foo()
```

```
class A {
 public:
  virtual void Foo();
};

class B : public A {
 public:
  virtual void Foo();
};

class C : public B {
};

class D : public C {
 public:
  virtual void Foo();
};

class E : public C {
};
```

Next time:

- How dynamic dispatch works
- Static dispatch
- Abstract classes
- Polymorphism with constructor, destructors & STL
- C++ casting