Fork & Pipe

Computer Systems Programming, Spring 2023

Instructor: Travis McGaha

TAs:

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

Logistics

- HW3 Posted Due Thursday 3/30 @ 11:59
  - Should have everything you need
  - Should be on the shorter side theoretically

- Check-in 07 to be released today/tomorrow
  - Due Before Monday’s lecture

- Project Partner Sign up to be released soon

Lecture Summary

- A `unique_ptr` ***takes ownership*** of a pointer
  - Cannot be copied, but can be moved
  - `get()` returns a copy of the pointer, but is dangerous to use; better to use `release()` instead
  - `reset()` `delete`s old pointer value and stores a new one
- A `shared_ptr` allows shared objects to have multiple owners by doing *reference counting*
  - `delete`s an object once its reference count reaches zero
- A `weak_ptr` works with a shared object but doesn't affect the reference count
  - Can't actually be dereferenced, but can check if the object still exists and can get a `shared_ptr` from the `weak_ptr` if it does

Some Important Smart Pointer Methods

Visit http://www.cplusplus.com/ for more information on these!

- `std::unique_ptr U;`
  - `U.get()` Returns the raw pointer U is managing
  - `U.release()` U stops managing its raw pointer and returns the raw pointer
  - `U.reset(q)` U cleans up its raw pointer and takes ownership of q
- `std::shared_ptr S;`
  - `S.get()` Returns the raw pointer S is managing
  - `S.use_count()` Returns the reference count
  - `S.unique()` Returns true iff S.use_count() == 1
- `std::weak_ptr W;`
  - `W.lock()` Constructs a shared pointer based off of W and returns it
  - `W.use_count()` Returns the reference count
  - `W.expired()` Returns true iff W is expired (W.use_count() == 0)

"Smart" Pointers

- Smart pointers still don't know everything, you must be careful with what pointers you give it to manage.
  - Smart pointers can't tell if a pointer is on the heap or not.
    - Still uses delete on default.
  - Smart pointers can't tell if you are re-using a raw pointer.

Using a non-heap pointer

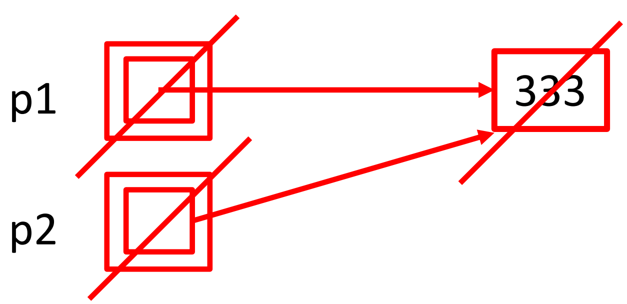
```
#include <cstdlib>
#include <memory>

using std::shared_ptr;
using std::weak_ptr;

int main(int argc, char **argv) {
  int x = 333;

  shared_ptr<int> p1(&x);

  return EXIT_SUCCESS;
}
```

- Smart pointers can’t tell if the pointer you gave points to the heap!
  - Will still call delete on the pointer when destructed.

Re-using a raw pointer

```
#include <cstdlib>
#include <memory>

using std::unique_ptr;


int main(int argc, char **argv) {
  int *x = new int(333);

  unique_ptr<int> p1(x);

  unique_ptr<int> p2(x);

  return EXIT_SUCCESS;
}
```

- Smart pointers can't tell if you are re-using a raw pointer.

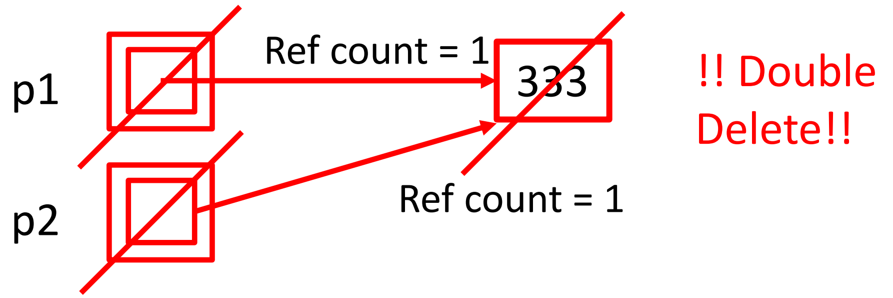


!! Double Delete!!

Re-using a raw pointer

```
#include <cstdlib>
#include <memory>

using std::shared_ptr;

int main(int argc, char **argv) {
  int *x = new int(333);

  shared_ptr<int> p1(x);  // ref count:

  shared_ptr<int> p2(x);  // ref count:

  return EXIT_SUCCESS;
}
```

- Smart pointers can't tell if you are re-using a raw pointer.



Re-using a raw pointer: Fixed Code

```
#include <cstdlib>
#include <memory>

using std::shared_ptr;

int main(int argc, char **argv) {
  int *x = new int(333);

  shared_ptr<int> p1(new int(333));

  shared_ptr<int> p2(p1); // ref count:

  return EXIT_SUCCESS;
}
```

- Smart pointers can't tell if you are re-using a raw pointer.
  - Takeaway: be careful!!!!
  - Safer to use cctor
  - To be extra safe, don't have a raw pointer variable!

Lecture Outline

- **`fork()` and `wait()`**
- stdin, stdout, and the file table
- `exec*()` and `pipe()`
- HW4 Overview & Hints

Poll Everywhere

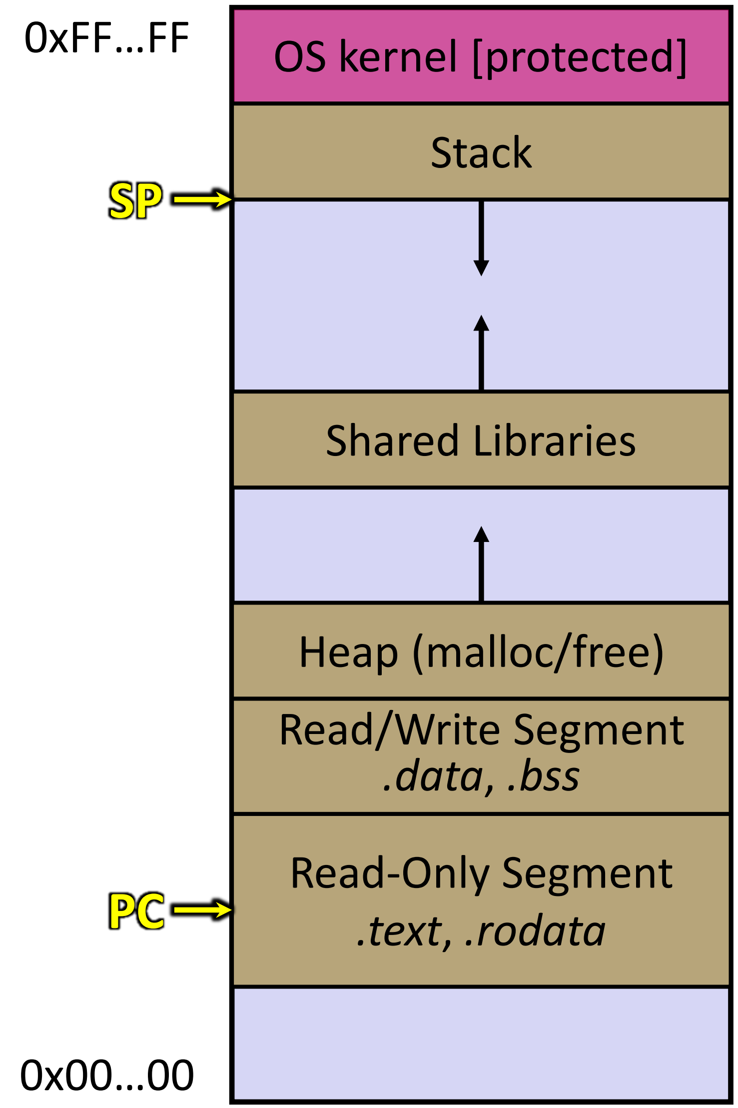
pollev.com/tqm

- Any questions from the check-in before we begin?

Review: Address Spaces

- A process has its own *address space*
  - Includes segments for different parts of memory
  - A process usually has one or more threads
    - A thread tracks its current state using the stack pointer (SP) and program counter (PC)
- New processes are created with:

```
pid_t fork();
```

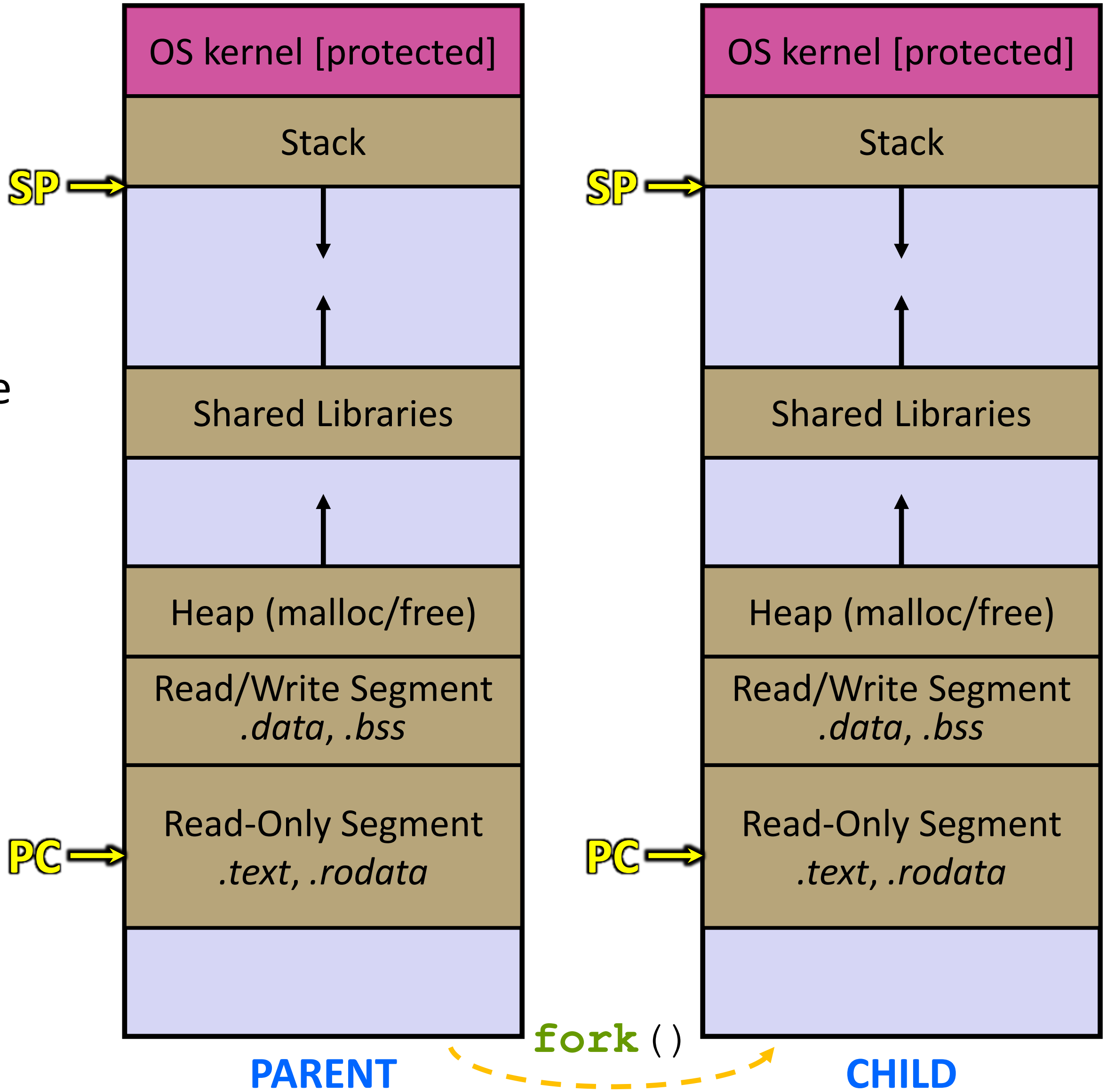


Creating New Processes

- `pid_t fork();`
  - Creates a new process (the "child") that is an *exact clone** of the current process (the "parent")
    - *almost everything
  - The new process has a separate virtual address space from the parent

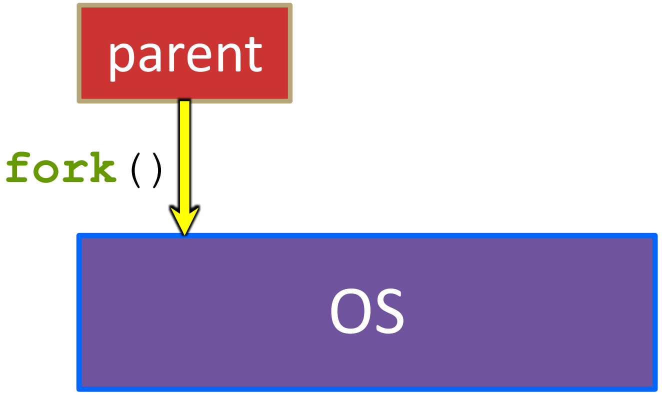
`fork()` and Address Spaces

- Fork causes the OS to clone the address space
  - The *copies* of the memory segments are (nearly) identical
  - The new process has *copies* of the parent's data, stack-allocated variables, open file descriptors, etc.



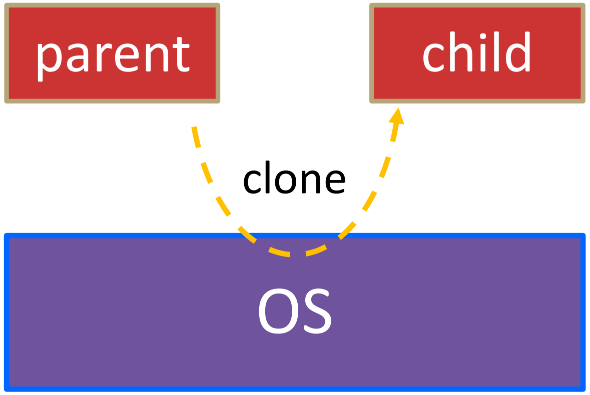
`fork()`

- `fork()` has peculiar semantics
  - The parent invokes `fork()`
  - The OS clones the parent
  - *Both* the parent and the child return from fork
    - Parent receives child's pid
    - Child receives a 0



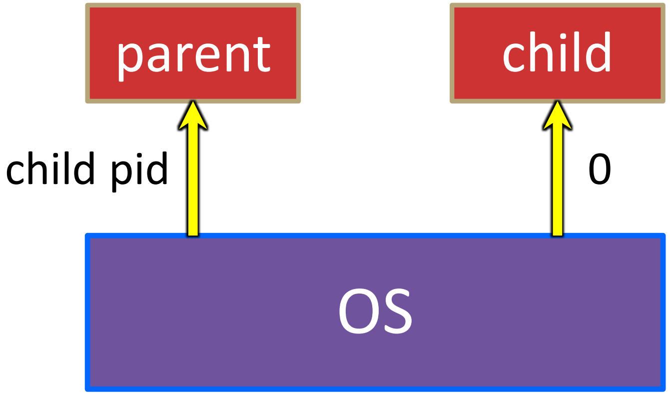
`fork()`

- `fork()` has peculiar semantics
  - The parent invokes `fork()`
  - The OS clones the parent
  - *Both* the parent and the child return from fork
    - Parent receives child's pid
    - Child receives a 0



`fork()`

- `fork()` has peculiar semantics
  - The parent invokes `fork()`
  - The OS clones the parent
  - *Both* the parent and the child return from fork
    - Parent receives child's pid
    - Child receives a 0



fork() example

```
cout << "Hello!" << endl;
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 595;
} else {
  x = 593;
}
cout << x << endl;
```

Always prints "Hello"

fork() example

```
cout << "Hello!" << endl;
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 595;
} else {
  x = 593;
}
cout << x << endl;
```

Always prints "Hello"

fork() example

Parent Process (PID = X)

```
cout << "Hello!" << endl;
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 595;
} else {
  x = 593;
}
cout << x << endl;
```

fork_ret = Y

Child Process (PID = Y)

```
cout << "Hello!" << endl;
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 595;
} else {
  x = 593;
}
cout << x << endl;
```

fork_ret = 0

fork()

Always prints "Hello"

Does NOT print "Hello"

fork() example

Parent Process (PID = X)

```
cout << "Hello!" << endl;
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 595;
} else {
  x = 593;
}
cout << x << endl;
```

fork_ret = Y

fork()

Child Process (PID = Y)

```
cout << "Hello!" << endl;
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 595;
} else {
  x = 593;
}
cout << x << endl;
```

fork_ret = 0

Always prints "Hello"

Always prints "593"

Always prints "595"

Exiting a Process

- `void exit(int status);`
  - Causes the current process to exit normally
  - Automatically called by `main()` when main returns
  - Exits with a return status (e.g. EXIT_SUCCESS or EXIT_FAILURE)
    - This is the same int returned by `main()`
  - The exit status is accessible by the parent process with `wait()` or `waitpid()`.

"join"-ing a Process

- ```
  pid_t waitpid(pid_t pid, int *wstatus,
                        int options);
  ```
  - The "process equivalent" of `pthread_join()`
  - Calling process waits for a child process (specified by **`pid`**) to exit
    - Also cleans up the child process
  - Gets the exit status of child process through output parameter **`wstatus`**
  - **`options`** are optional, pass in `0` for default options in most cases
  - Returns process ID of child who was waited for or `-1` on error
- ```
  pid_t wait(int *wstatus);
  ```
  - Equivalent of waitpid, but waits for ANY child

Demo: `fork_example`

- See `fork_example.cc`
  - Brief code demo to see the various states of a process
    - Running
    - Zombie
    - Terminated
  - Makes use of `sleep()`, `waitpid()` and `exit()`!

Poll Everywhere

pollev.com/tqm

- We've briefly mentioned that it is *possible* to have two processes share information. How could this be done?

Lecture Outline

- `fork()` and `wait()`
- **stdin, stdout, and the file table**
- `exec*()` and `pipe()`
- HW4 Overview & Hints

stdout, stdin, stderr

- By default, there are three “files” open when a program starts
  - stdin: for reading terminal input typed by a user
    - `cin` in C++
    - `System.in` in Java
  - stdout: the normal terminal output.
    - `cout` in C++
    - `System.out` in Java
  - stderr: the terminal output for printing errors
    - `cerr` in C++
    - `System.err` in Java

stdout, stdin, stderr

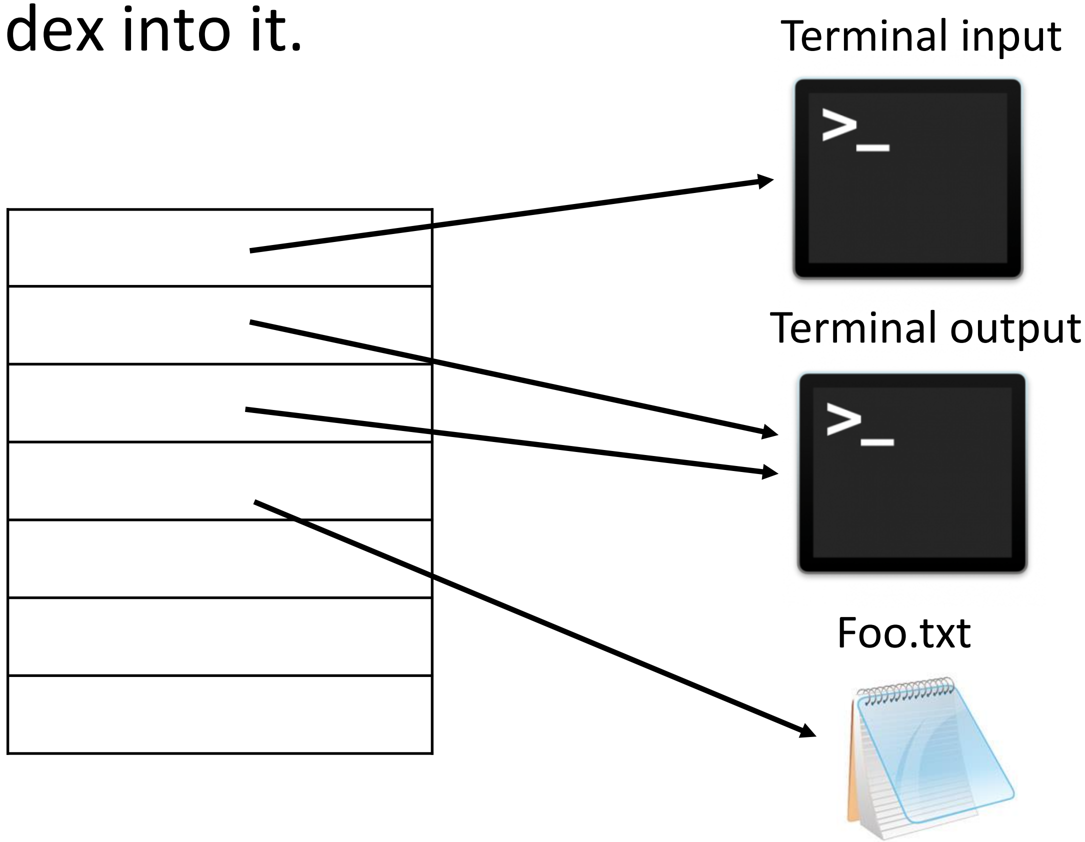
- stdin, stdout, and stderr all have initial file descriptors constants defined in `unistd.h`
  - `STDIN_FILENO   -> 0`
  - `STDOUT_FILENO  -> 1`
  - `STDERR_FILENO  -> 2`

- These will be open on default for a process

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

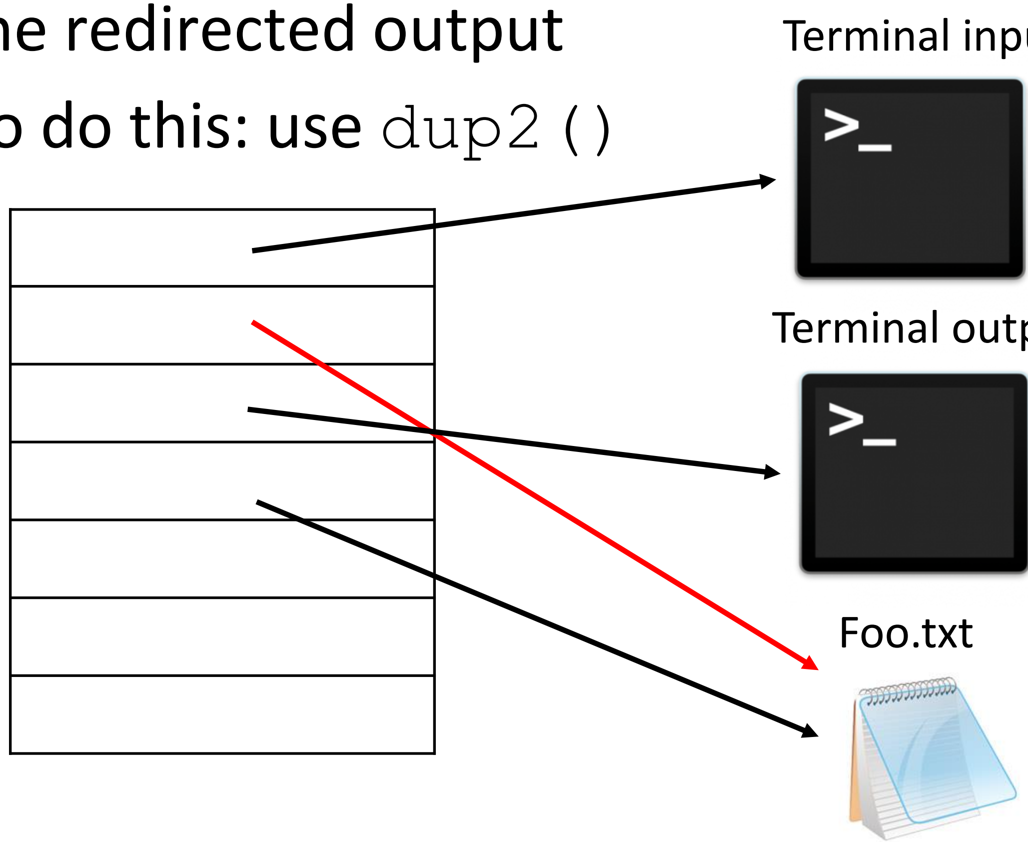
File Descriptor Table

- In addition to an address space, each process will have 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.



Redirecting stdin/out/err

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



Poll Everywhere

pollev.com/tqm

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

```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");
}
```

Lecture Outline

- `fork()` and `wait()`
- stdin, stdout, and the file table
- **`exec*()` and `pipe()`**
- HW4 Overview & Hints

exec*()

- Loads in a new program for execution

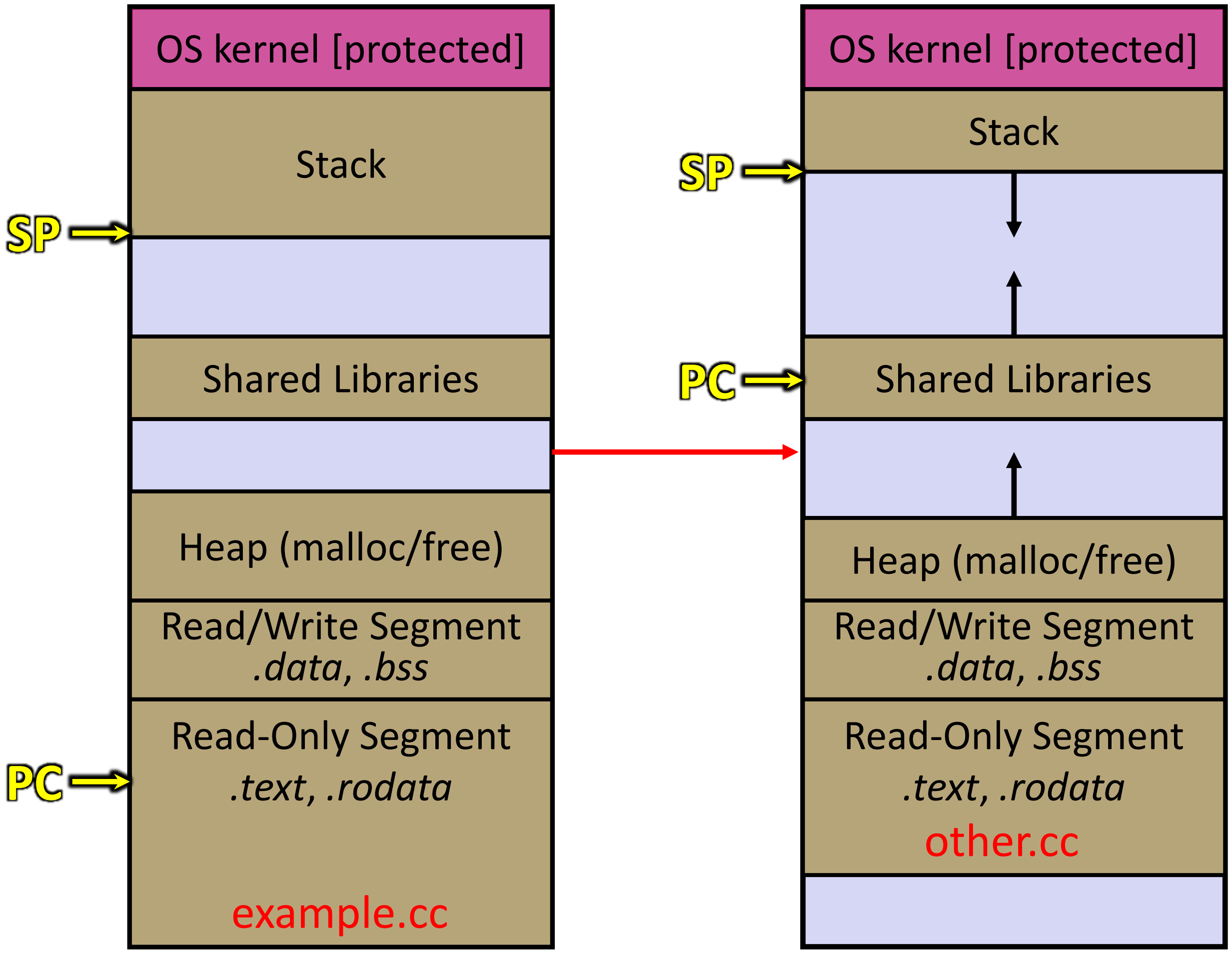
- PC, SP, registers, and memory are all reset so that the specified program can run

execvp()

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

Exec Visualization

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



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

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

Exec Demo

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

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

Pipes

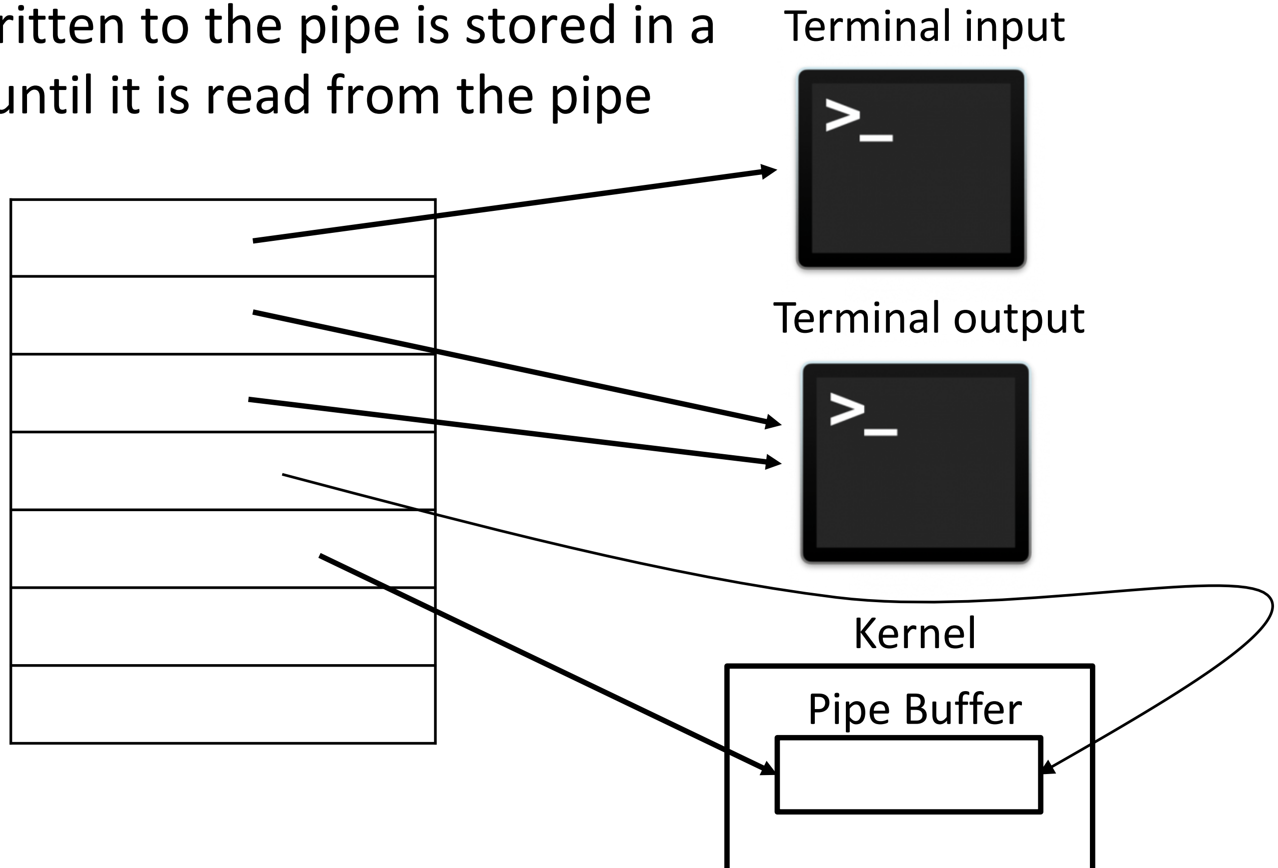
```
int pipe(int pipefd[2]);
```

- 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 open files (and pipes)
- Exec does NOT reset open files

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



Lecture Outline

- `fork()` and `wait()`
- stdin, stdout, and the file table
- `exec*()` and `pipe()`
- **HW4 Overview & Hints**

Unix Shell Commands

- Commands can also specify flags
  - E.g. "`ls -l`" lists the files in the specified directory in a more verbose format

- Revisiting the design philosophy:
  - Programs should "Do One Thing And Do It Well."
  - Programs should be written to work together
  - Write programs that handle text streams, since text streams is a universal interface.

- These programs can be easily combined with UNIX Shell operators to solve more interesting problems

Unix Shell Control Operators: Pipe

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

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

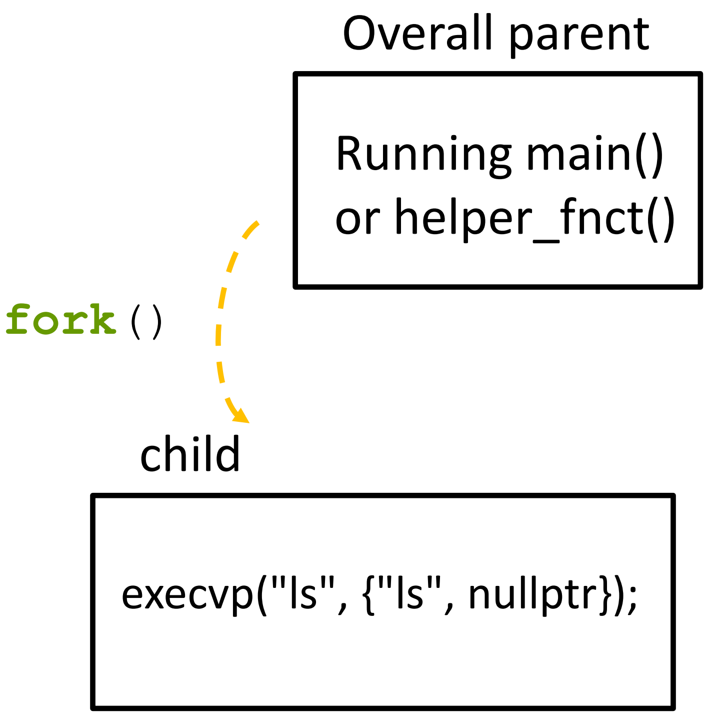
- Some sample programs provided to help with implementation ideas.

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"`

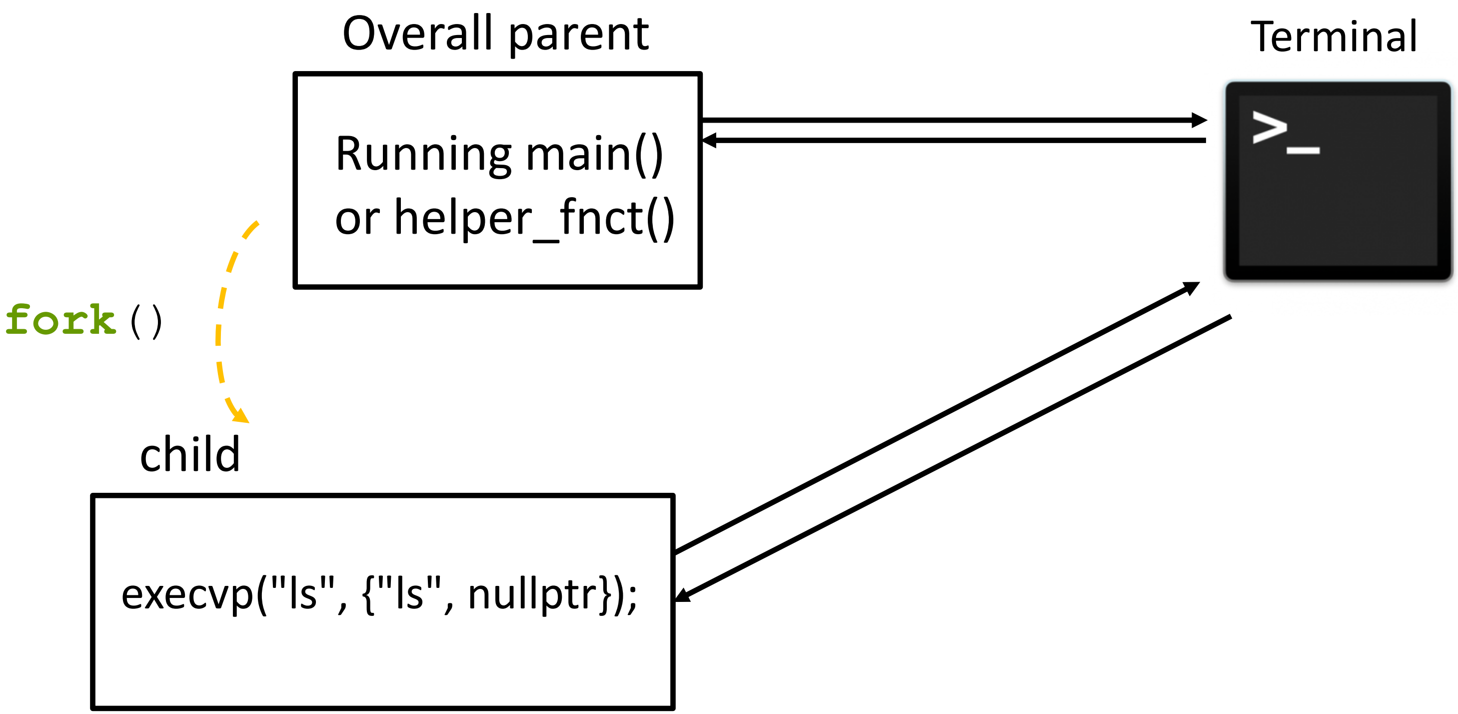
HW4 Example Line

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



HW4 Example Line

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



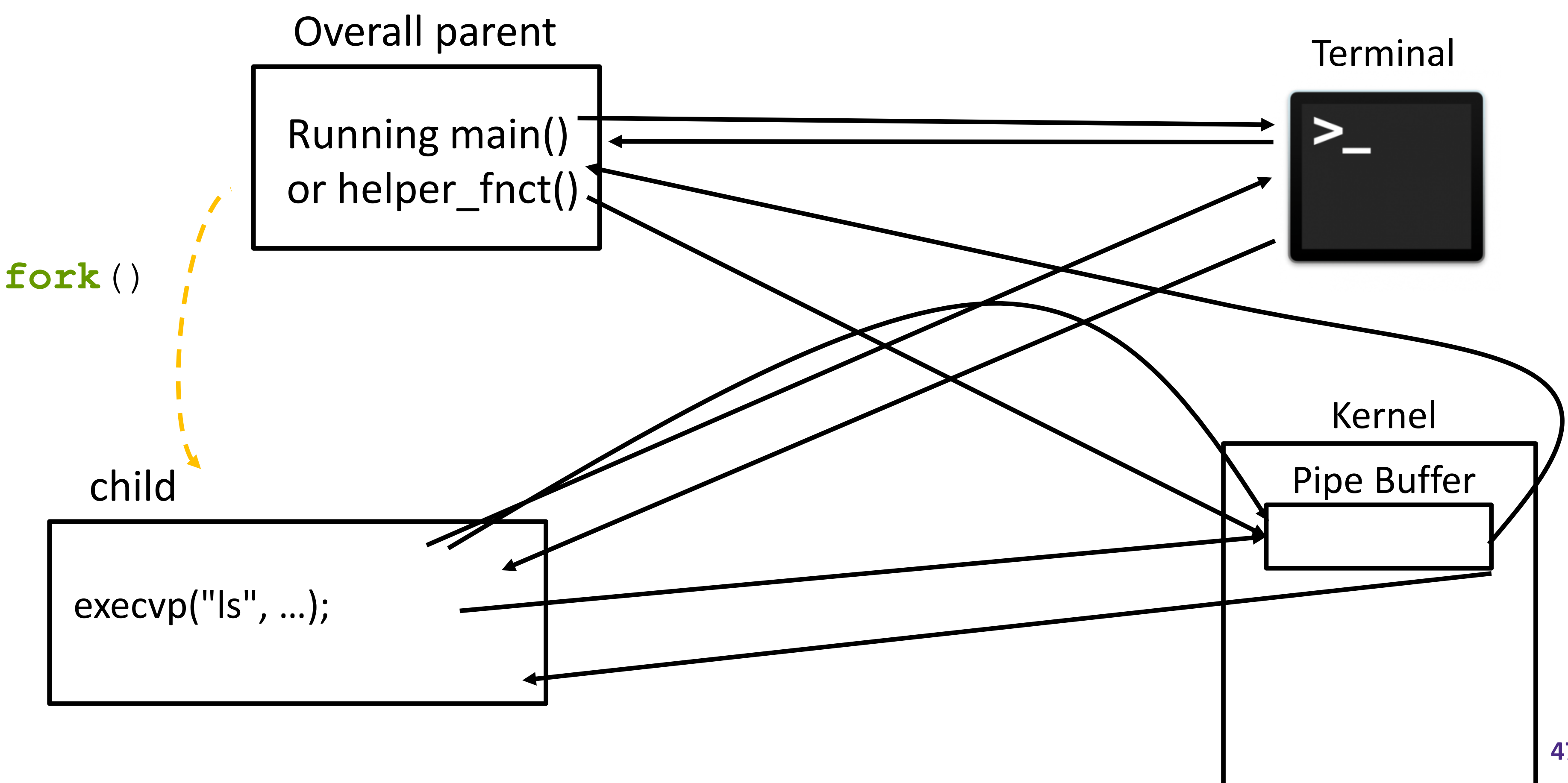
HW4 Hints

- 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

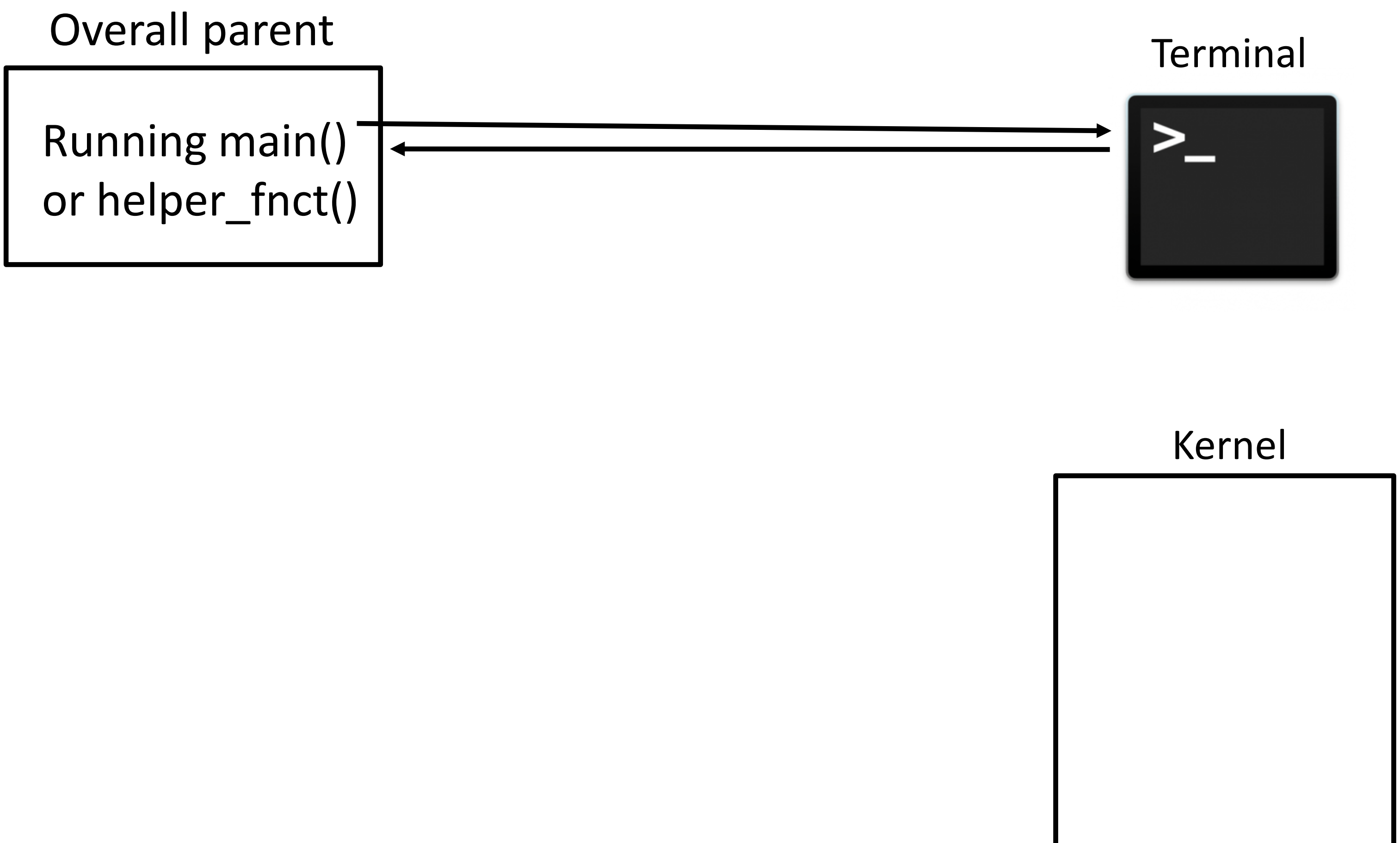
HW4 Example Line 1

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



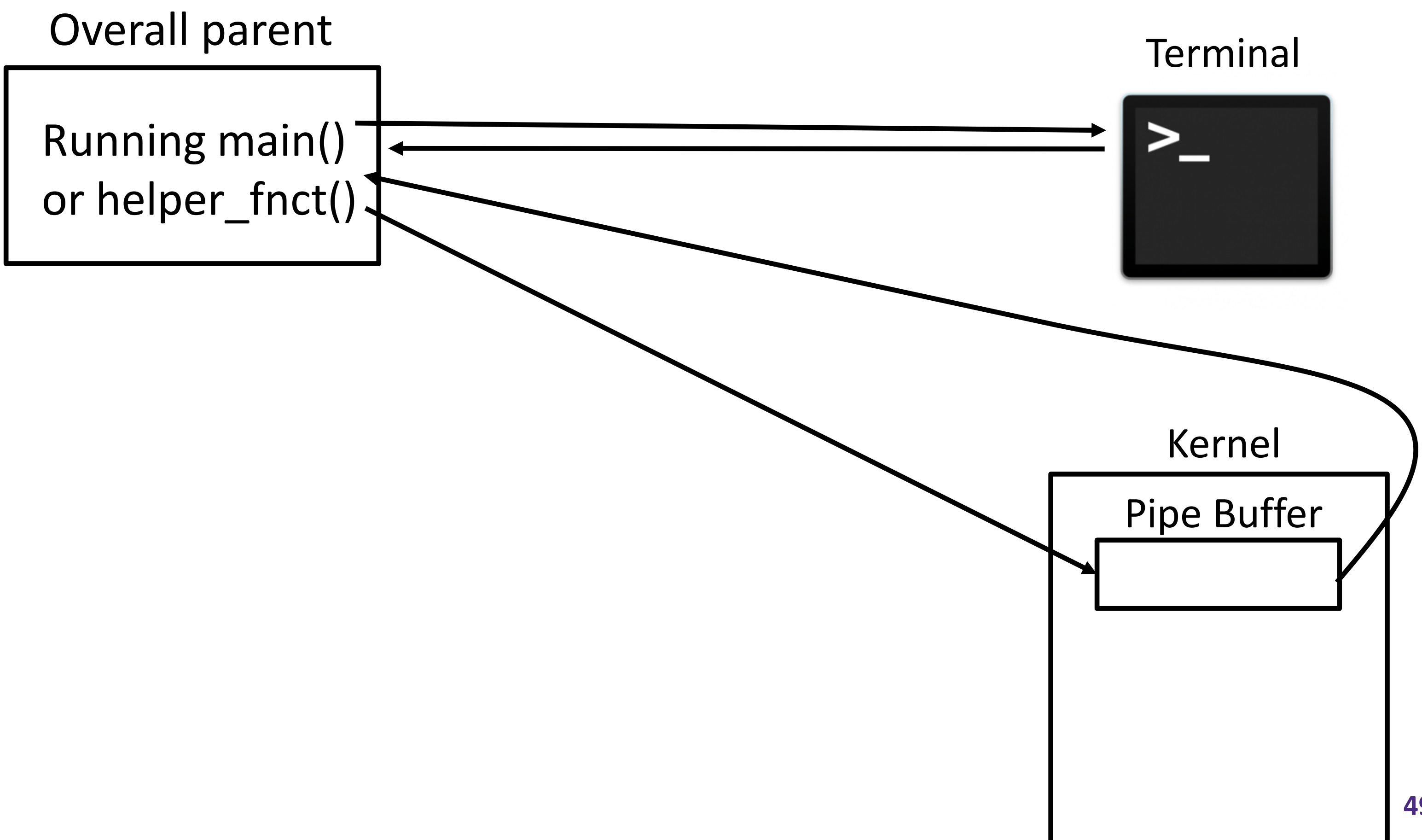
HW4 Example Line 1

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



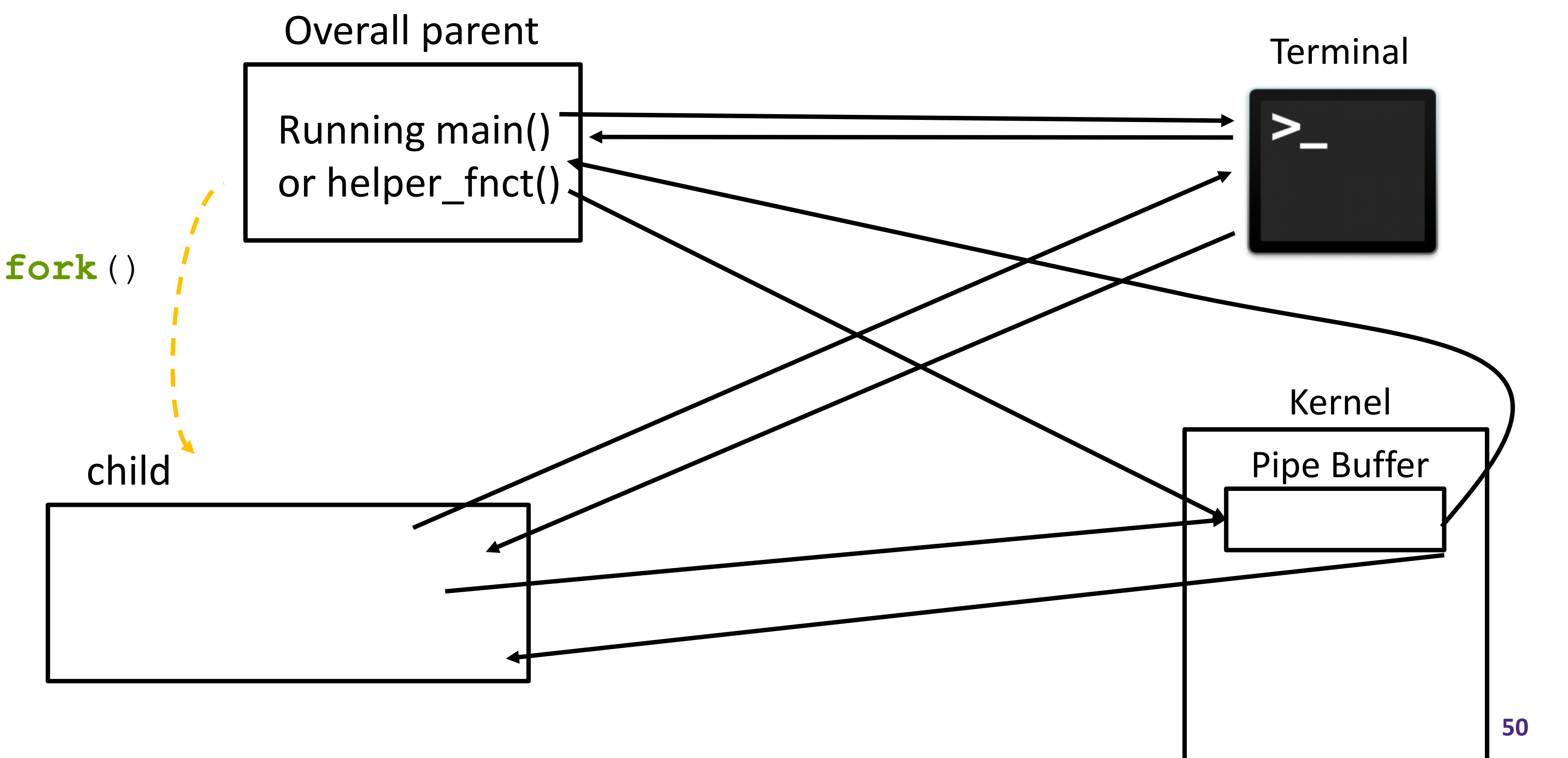
HW4 Example Line 1

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



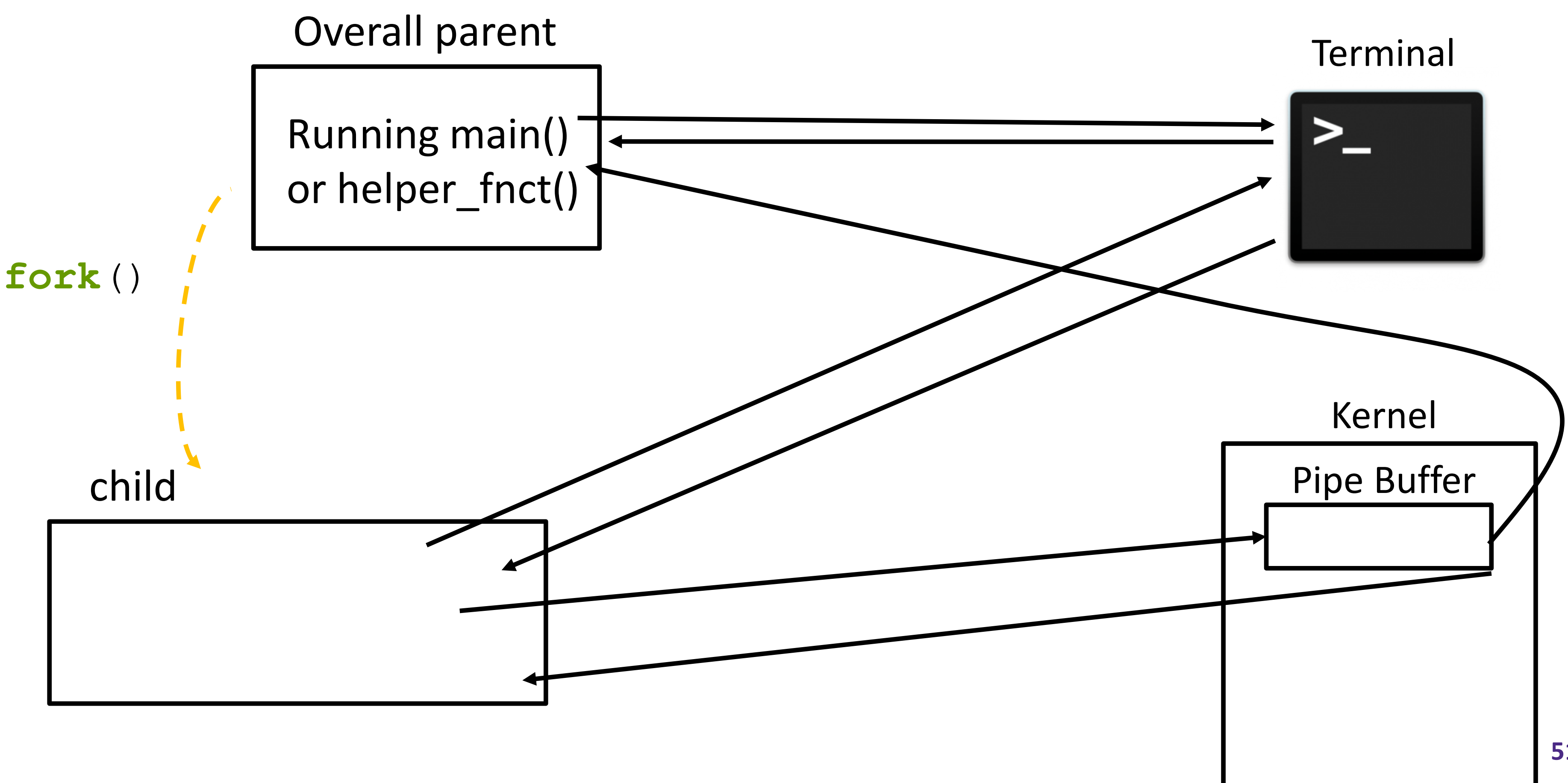
HW4 Example Line 1

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



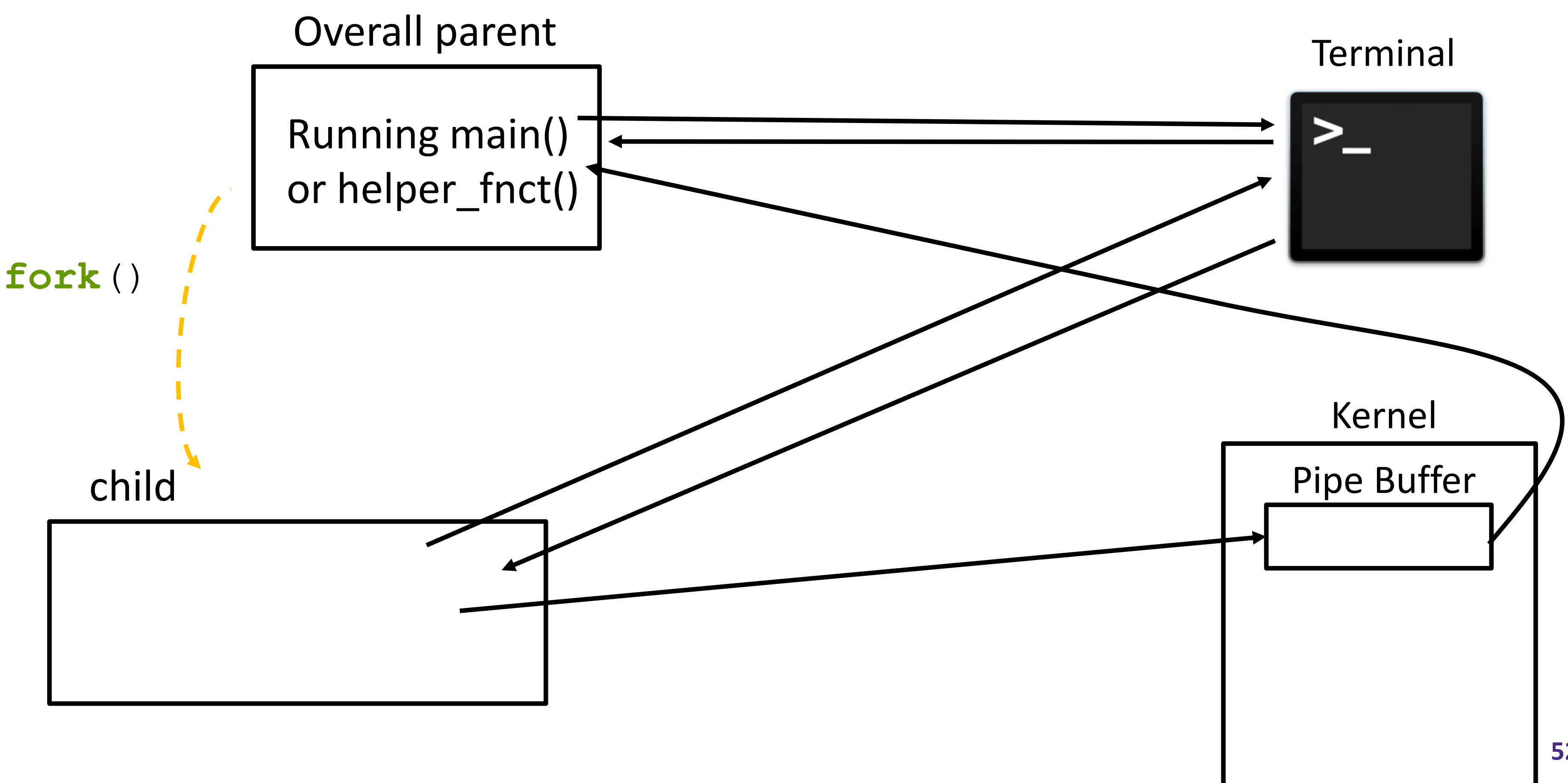
HW4 Example Line 1

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



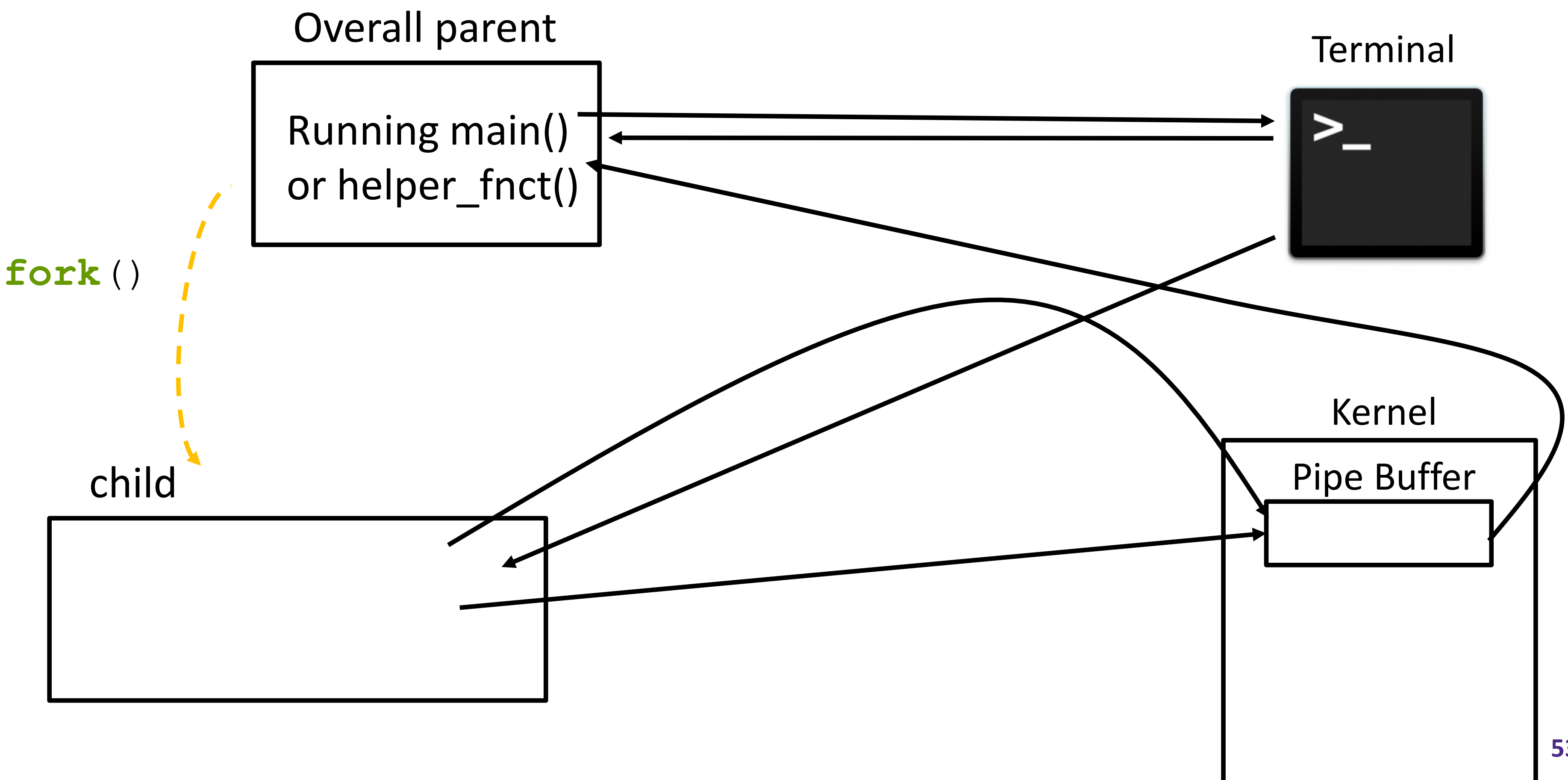
HW4 Example Line 1

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



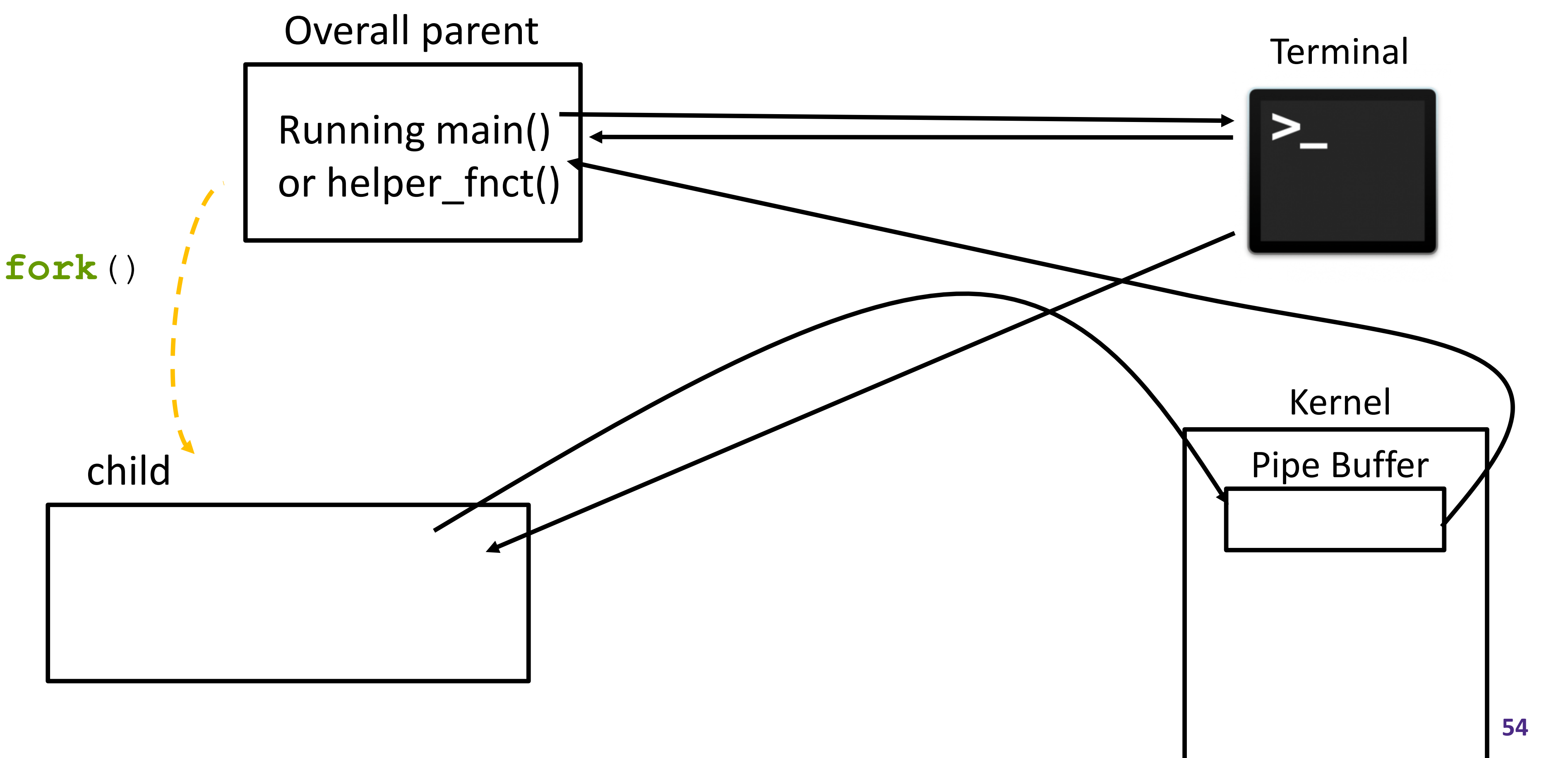
HW4 Example Line 1

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



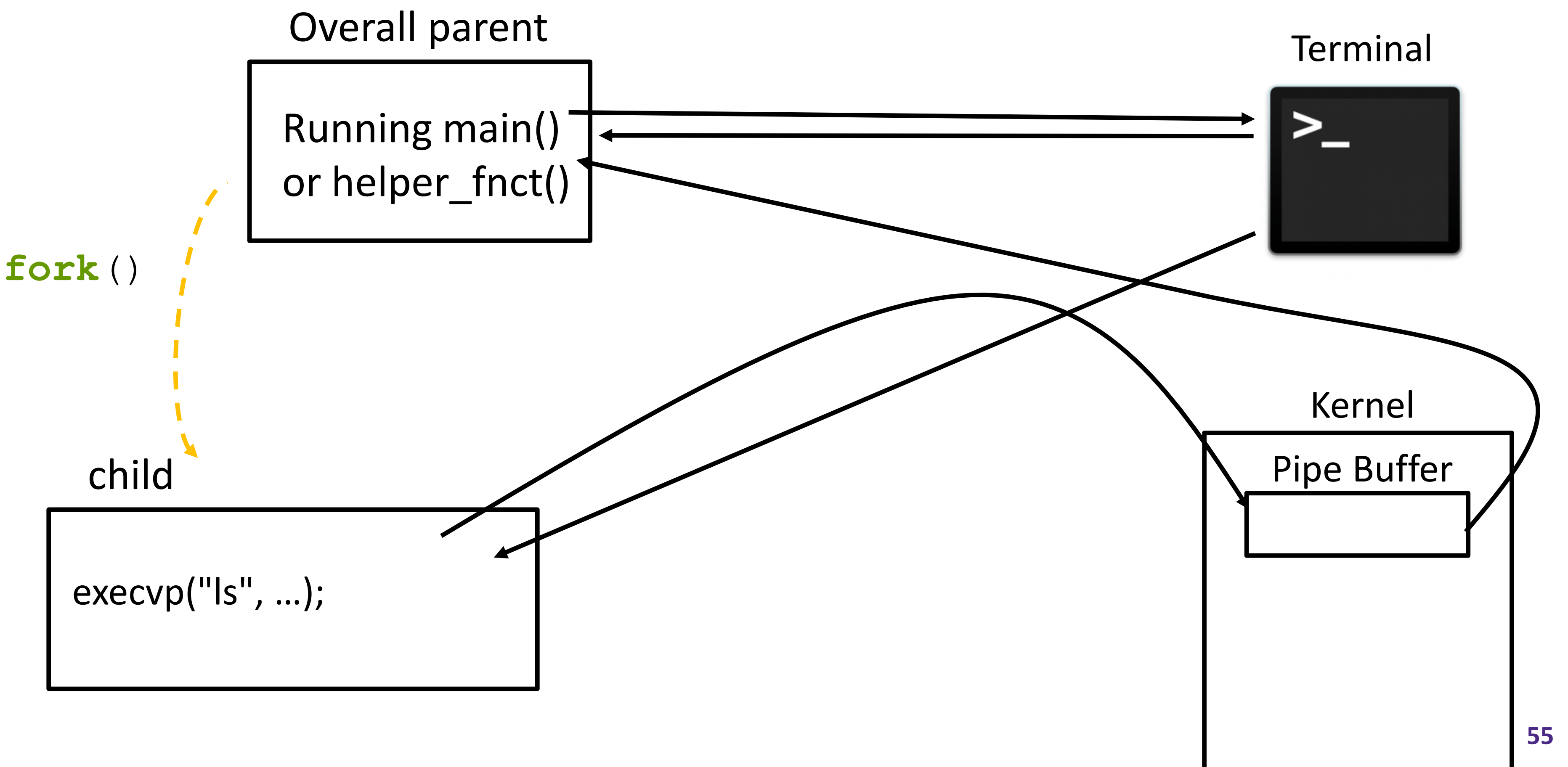
HW4 Example Line 1

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



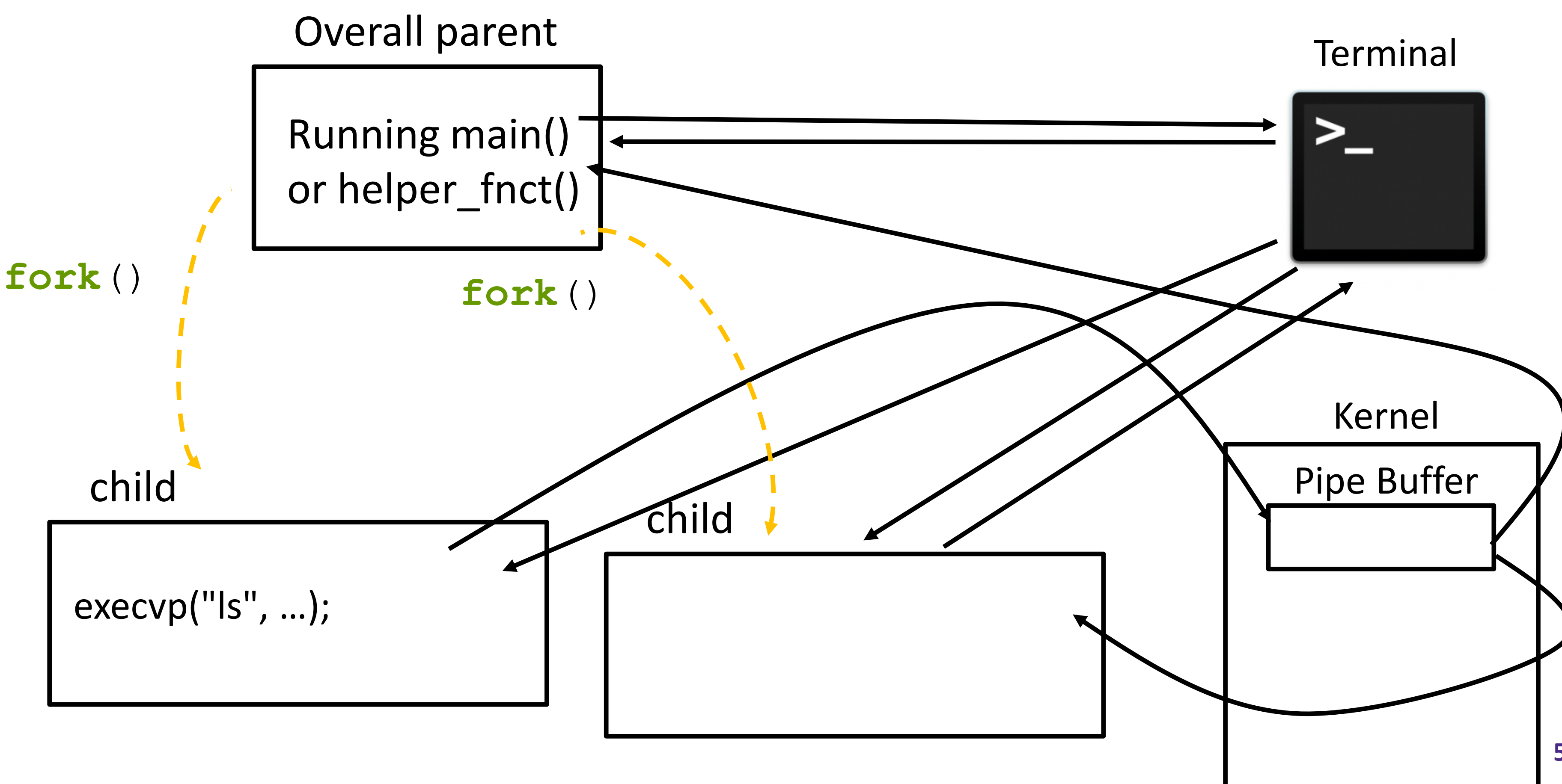
HW4 Example Line 1

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



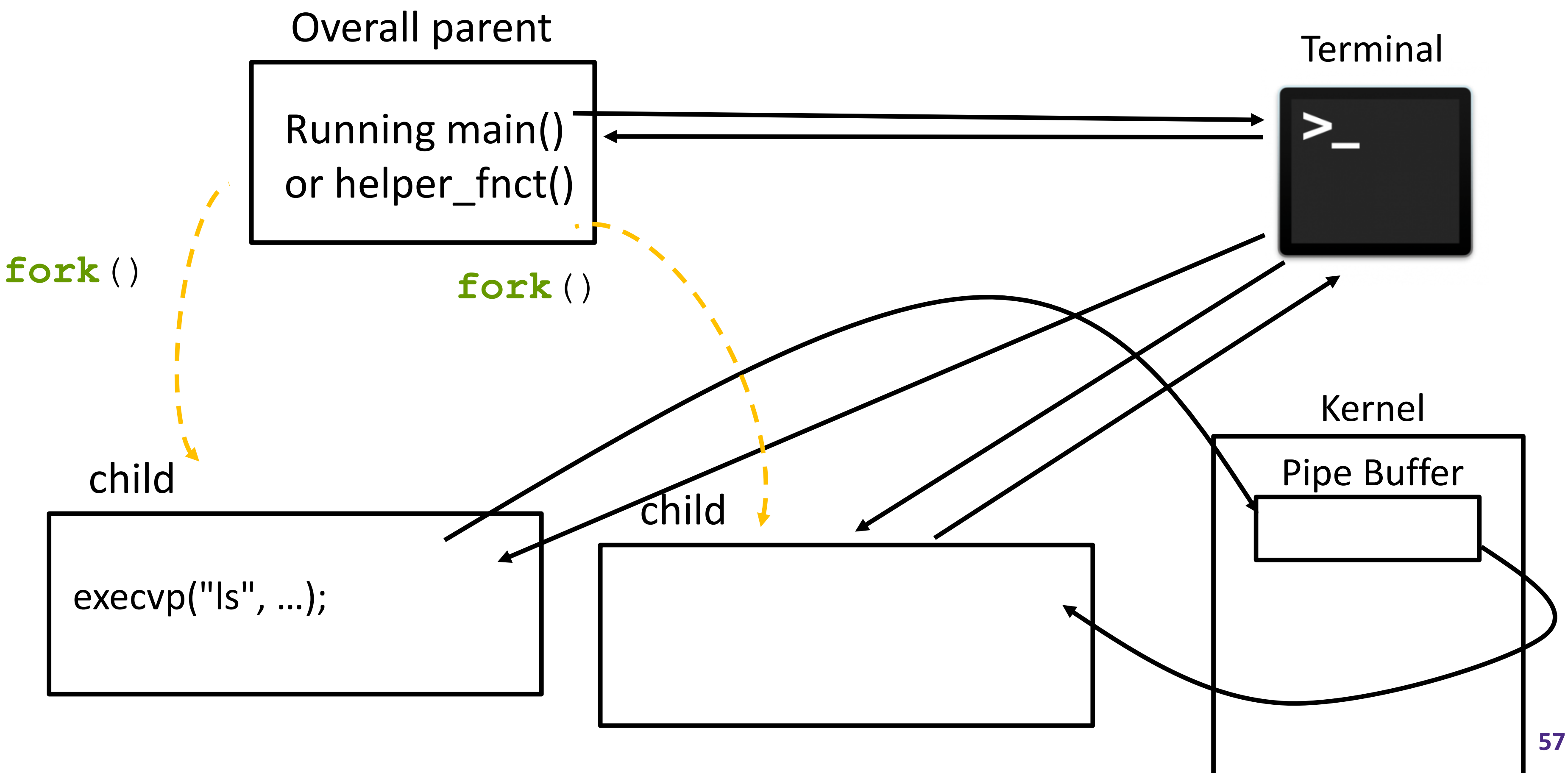
HW4 Example Line 1

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



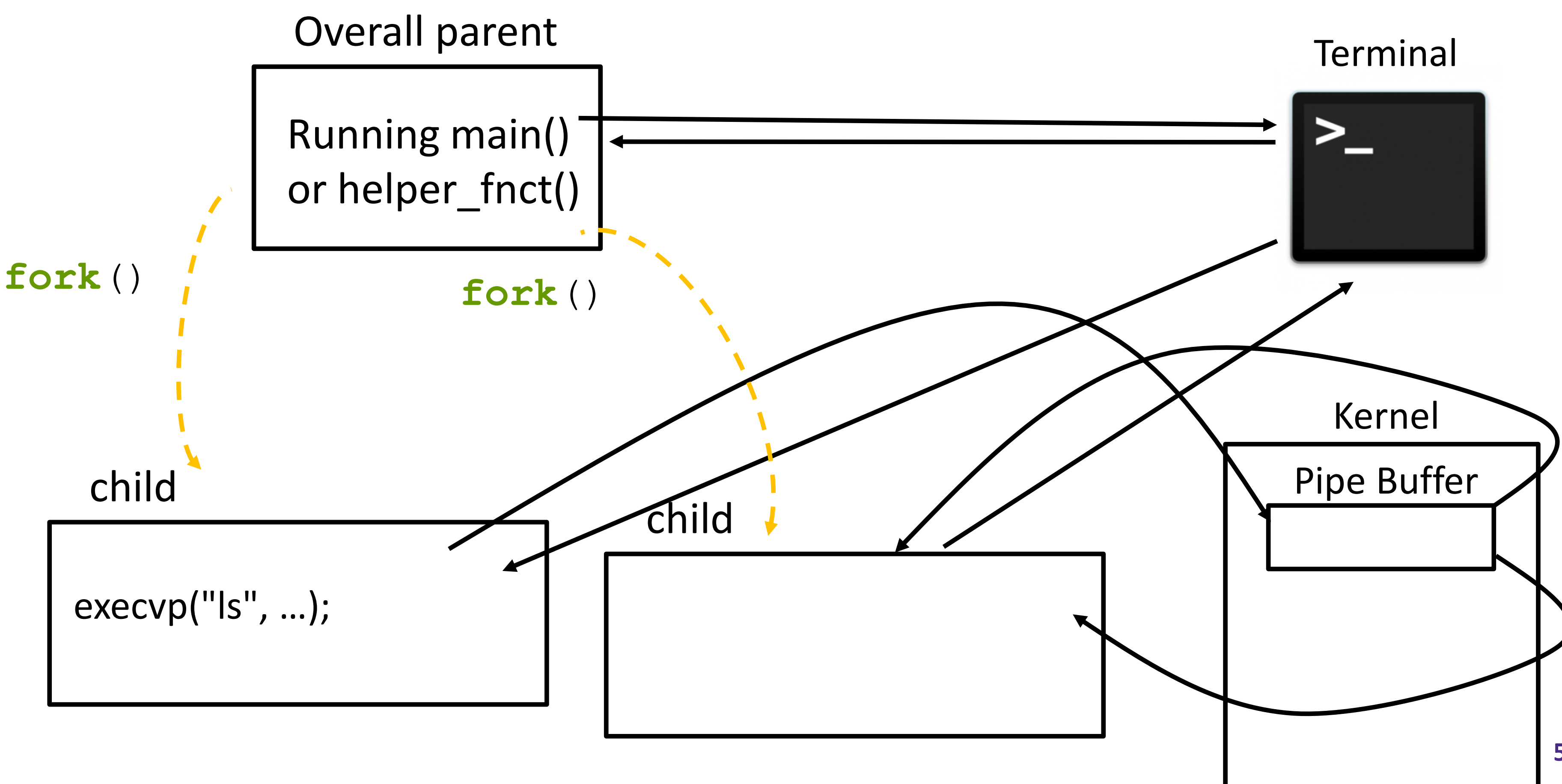
HW4 Example Line 1

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



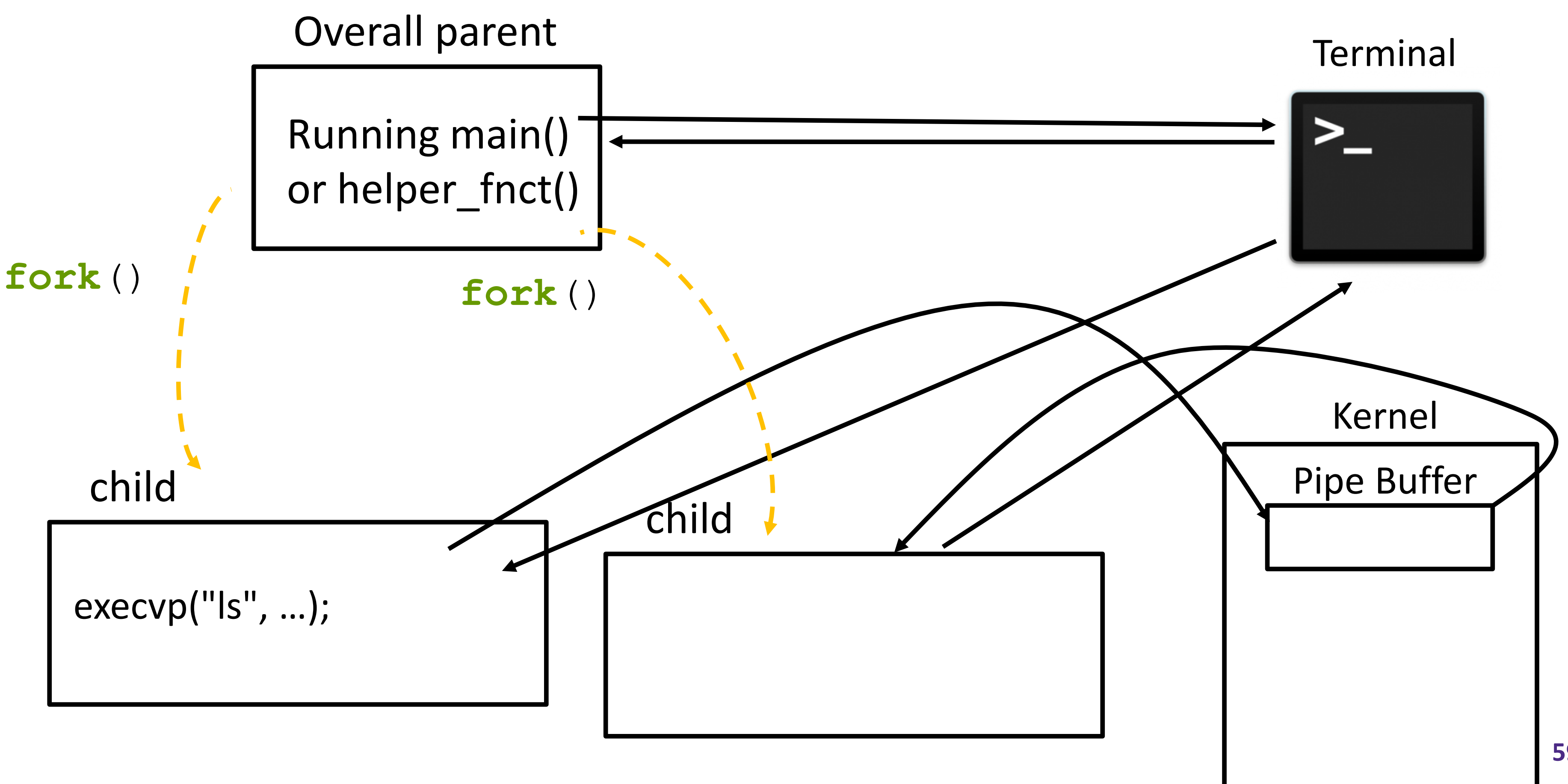
HW4 Example Line 1

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



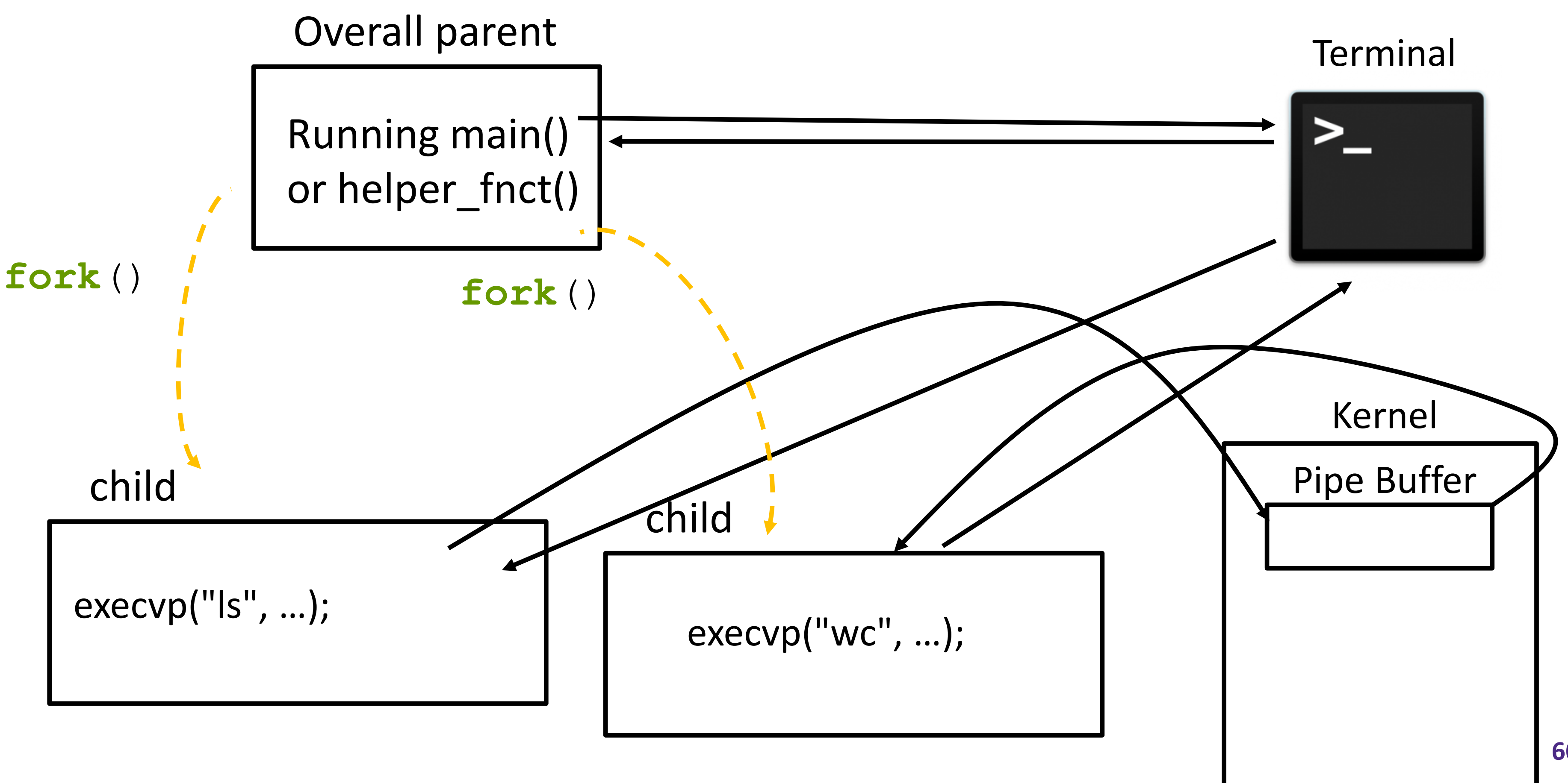
HW4 Example Line 1

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



HW4 Example Line 1

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



HW4 Example Line 2

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