CIT 5950
Recitation 10

Pipe() and HW4

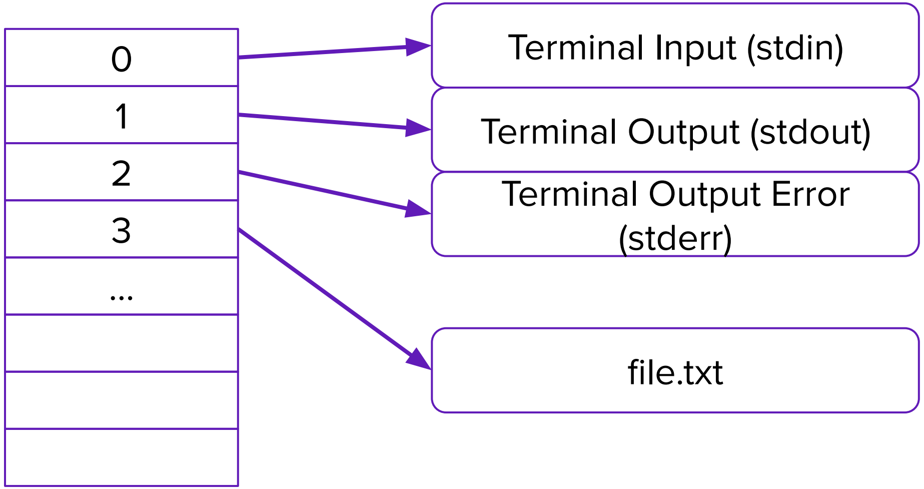
Logistics

- Project
  - Due May 1st, 11:59pm
- HW3
  - Due tomorrow at Midnight
- HW4
  - Released! Overview in this recitation
  - Due Friday April 26th, 11:59pm

File Descriptors, Redirections & Pipes

File Descriptor

- Unique id that refers to a file
- Type int
- read(2) and write(2)
- open(2) and close(2)
  - Open with unique permissions
  - Read only, write only, read&write, etc
- **Each process has unique file descriptor table**
- 0, 1, 2 reserved for stdin, stdout, stderr



0

1

2

3

...

Terminal Input (stdin)

Terminal Output (stdout)

Terminal Output Error (stderr)

file.txt

Quick Example

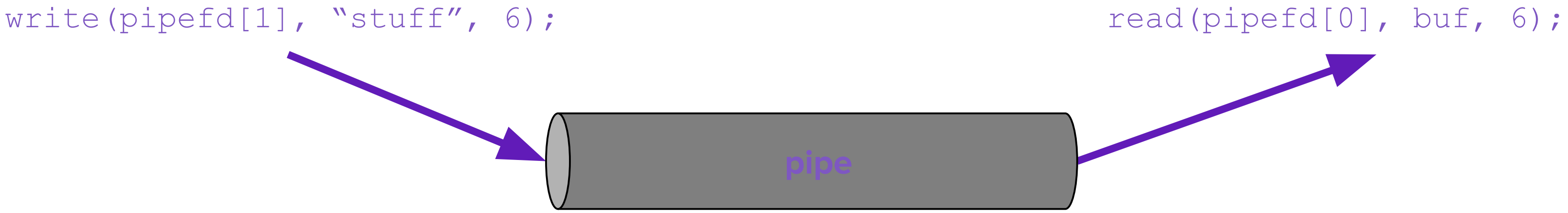
- `read(STDIN_FILENO, buf, 30);`
  - Reads from terminal input and stores to buffer
- `write(STDERR_FILENO, "error message\n", 15);`
  - Write to terminal output error
- `write(STDIN_FILENO, "trying to write\n", 17);`
  - Error. STDIN is “read only”

Redirections

- Redirect a file descriptor to point to some other file!
- dup2(int oldfd, int newfd)
    - Whatever file that was pointed to by **oldfd** is now pointed to file pointed to by **newfd**
- `dup2(newfd, STDIN_FILENO)`
    - Redirect STDIN to newfd. What does this mean?
    - Anything that was supposed to be read from stdin, which was terminal input, will come from newfd
- `dup2(newfd, STDOUT_FILENO)`
    - Redirect STDOUT to newfd. What does this mean?
    - Anything that was supposed to be outputted to stdout, will now be outputted to newfd

Pipes

- FIFO data structure with a read end and write end
  - Picture a pipe with water flowing into (write end) and out of (read end)
- pipe(2) system call. `pipe(int pipefd[2])`
  - Creates the pipe data structure pointed to by pipefd
  - `pipefd[0]` = read-end, `pipefd[1]` = write-end

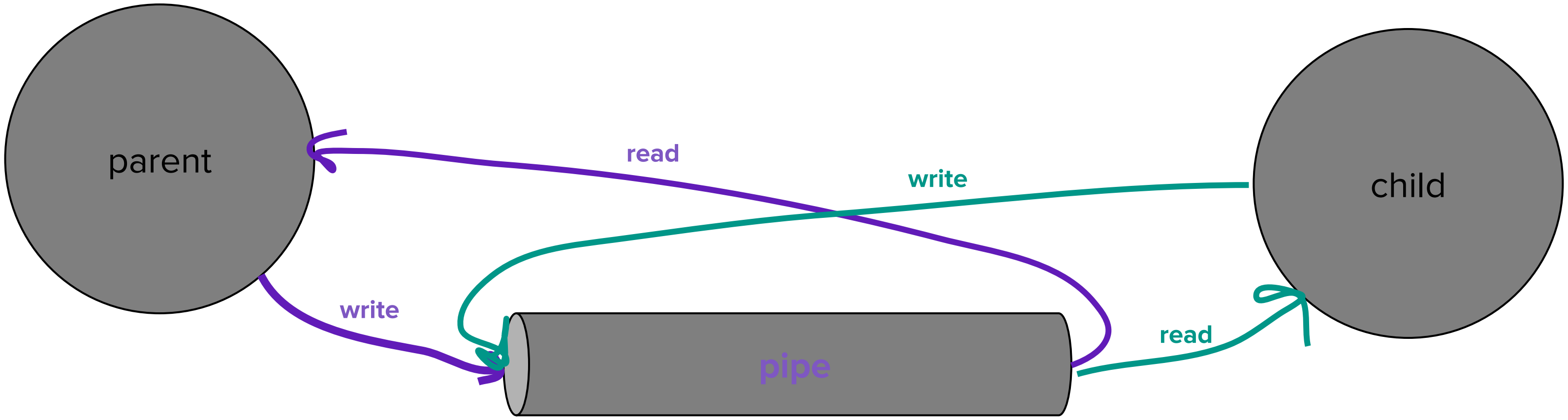


```
write(pipefd[1], "stuff", 6);
```

```
read(pipefd[0], buf, 6);
```

Pipes and processes

- File Descriptor table is “shared” among processes
    - → pipes are shared!!!!
- Child processes has its own copy of each pipe end



Some tips

DRAW! It is easier to visualize what points where.

READ! The system calls related to file descriptors. open(2), close(2), dup2(2), pipe(2)

READ CAREFULLY! The man pages for above. Really know what’s going on.

E.g. What happens when we fork(2) after pipe(2)?

What happens if we close(2) in a child process?

Processes and files/pipes

- If we create a pipe or access a file, there is one instance of it system wide
- When a process forks, it copies the file descriptors of the parent
- Multiple process can have access to the same file/pipe, but through their own file descriptors.
- When one process closes its file descriptors, other processes file descriptors remain open

dup2() and redirection

We can use dup2() to redirect a file descriptor to something else.

Each process also has its own file descriptors tables.

Fork copies the file descriptor of the parent into the child.

Always writes

```
child
parent
```

To the file “hello.txt”

```
int main(int argc, char* argv[]) {
  int fd = open("hello.txt", O_RDWR);
  pid_t pid =  fork();
  if (pid == 0) {
     wrapped_write(fd, "child"); // helper function to write a string to a fd
     close(fd);
     exit(EXIT_SUCCESS);
  }
  waitpid(pid, nullptr, 0);
  dup2(fd, STDOUT_FILENO);  // redirects STDOUT to the file specified by fd
  cout << "parent\n";  // writes to STDOUT_FILENO
}
```

Exercise 1

dup2 Exercise 1

```
int main(int argc, char* argv[]) {
  int fd = open("antennas.txt", O_RDWR);
  pid_t pid =  fork();
  close(STDOUT_FILENO);
  if (pid == 0) {
     cout << "storm\n";
     dup2(fd, STDOUT_FILENO);
     cout << "static\n";
     exit(EXIT_SUCCESS);
  }
  waitpid(pid, nullptr, 0);
  cout << "sleep\n";
}
```

What is printed to the terminal and what is written to antennas.txt?

dup2 Exercise 1 Solution

```
int main(int argc, char* argv[]) {
 int fd = open("antennas.txt", O_RDWR);
 pid_t pid =  fork();
 close(STDOUT_FILENO);
 if (pid == 0) {
   cout << "storm\n";
   dup2(fd, STDOUT_FILENO);
   cout << "static\n";
   exit(EXIT_SUCCESS);
 }
 waitpid(pid, nullptr, 0);
 cout << "sleep\n";
}
```

antennas.txt contains:

static

what was printed:

Nothing gets printed to the terminal since `STDOUT_FILENO` has been closed for both parent and child

What is printed to the terminal and what is written to antennas.txt?

dup2 Exercise 1 Solution

```
int main(int argc, char* argv[]) {
  int fd = open("antennas.txt", O_RDWR);
  pid_t pid =  fork();
  close(STDOUT_FILENO);
  if (pid == 0) {
     cout << "storm\n";
     dup2(fd, STDOUT_FILENO);
     cout << "static\n";
     exit(EXIT_SUCCESS);
  }
  waitpid(pid, nullptr, 0);
  cout << "sleep\n";
}
```

closes STDOUT

redirects STDOUT to the file specified by fd

For `dup2(newfd, oldfd)`

newfd must be a valid, open file descriptor.

oldfd does not need to be open; if it is, dup2 will close it without complaining. If it's not already open, dup2 will just assign it the file descriptor newfd.

dup2 Exercise 1 Solution

```
int main(int argc, char* argv[]) {
  int fd = open("begin.txt", O_RDWR);
  pid_t pid =  fork();
  if (pid == 0) {
     dup2(STDOUT_FILENO, fd);  ←  fd points to cout
     wrapped_write(fd, "dust");
     cout << "crusader\n";
     close(STDOUT_FILENO);
     exit(EXIT_SUCCESS);
  }
  dup2(fd, STDOUT_FILENO);  ←  STDOUT and fd point to the same thing (begin.txt)
  cout << "star\n";
  close(fd);
  waitpid(pid, nullptr, 0)
  cout << "platinum\n";  ←  though we closed fd, STDOUT still points to file
}
```

dup2 Exercise 1 Solution

```
int main(int argc, char* argv[]) {
  int fd = open("begin.txt", O_RDWR);
  pid_t pid =  fork();
  if (pid == 0) {
     dup2(STDOUT_FILENO, fd);
     wrapped_write(fd, "dust");
     cout << "crusader\n";
     close(STDOUT_FILENO);
     exit(EXIT_SUCCESS);
  }
  dup2(fd, STDOUT_FILENO);
  cout << "star\n";
  close(fd);
  waitpid(pid, nullptr, 0)
  cout << "platinum\n";
}
```

begin.txt contains:
star
platinum

what was printed:
dust
crusader

Bonus question: what do we know about the order of the words being printed/written?

Pipe()

- Unidirectional
    - If you want two processes to have bidirectional communication, you **must** make two pipes

- Ex: If you want to make a child process that will send info to its parent
    - Start with the parent process
    - Create your pipe array: **int arr[2];**
    - Call pipe: **pipe(arr);**
        - this creates a pipe in the kernel, and adds two file descriptors to your fd table
    - Fork your second process (the one you want the current process to communicate with)
        - **int pid = fork();**
    - Parent and child should close the ends that they do not use
        - Child close read: **if (pid == 0) { close(arr[0]); }**
        - Parent close write: **if (pid != 0) { close(arr[1]); }**
    - Once your child is done writing, it will call **close(arr[1]).** This ensures EOF is sent to the pipe to be read by the parent.

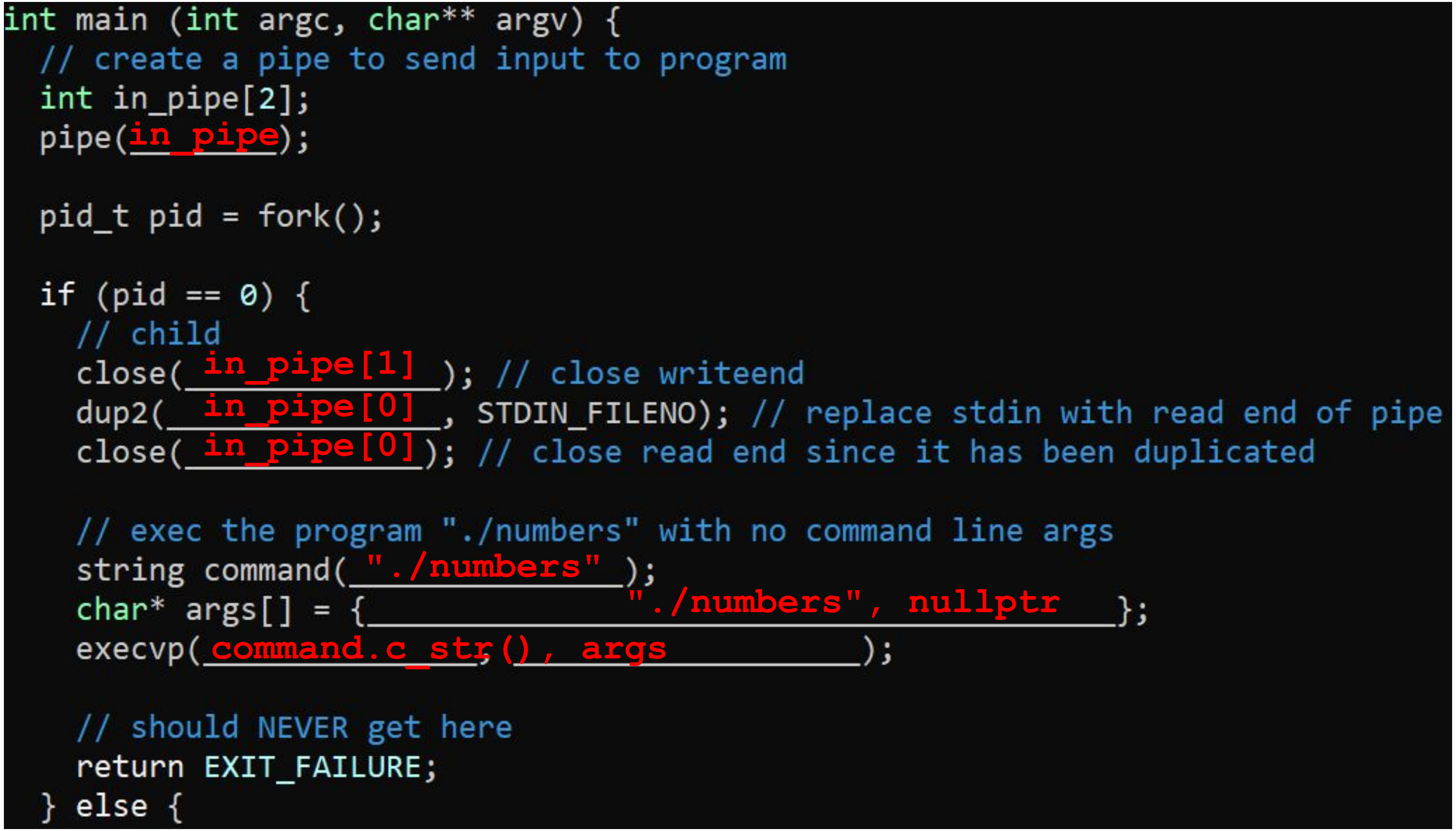
Exercise 2

Exercise: fill in the blanks

```
int main (int argc, char** argv) {
 // create a pipe to send input to program
 int in_pipe[2];
 pipe(       );
 pid_t pid = fork();
 if (pid == 0) { // child
   close(          ); // close writeend
   // replace stdin with read end of pipe
   dup2(          , STDIN_FILENO);
   // close read end since it has been duplicated
   close(          );
   string command(           ); // exec the program
"./numbers" with no command line args
   char* args[] = {                    };
   execvp(               ,    );
   return EXIT_FAILURE; // should NEVER get here
 }
```

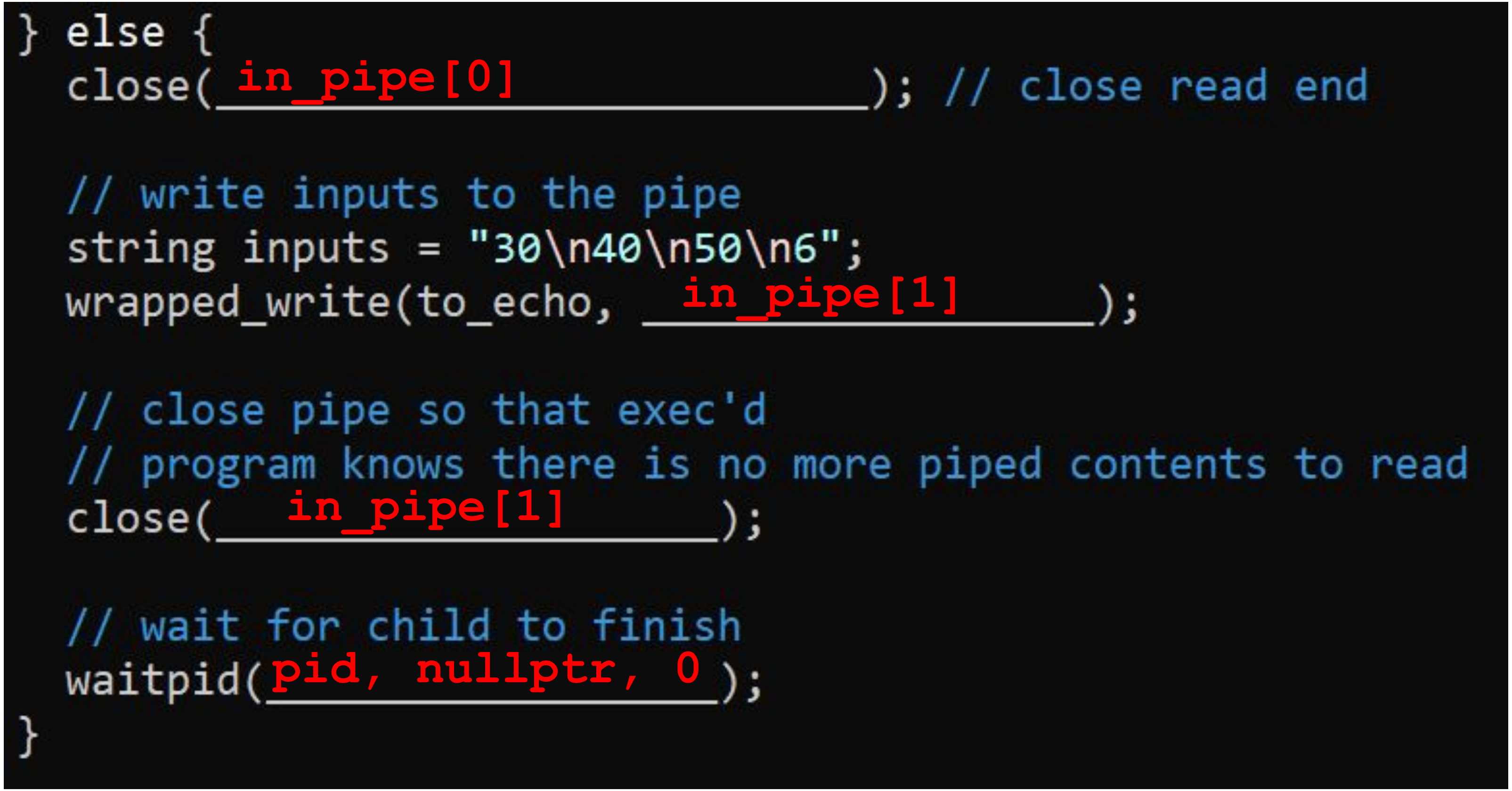
```
else {
   close(          ); // close read end

   // write inputs to the pipe
   string inputs = "30\n40\n50\n6";
   wrapped_write(to_echo,           );

   // close pipe so that exec'd program
   // knows there is no more piped contents to read
   close(          );

   // wait for child to finish
   waitpid(               );
  }
}
```

Exercise: fill in the blanks



```
int main (int argc, char** argv) {
  // create a pipe to send input to program
  int in_pipe[2];
  pipe(in_pipe);

  pid_t pid = fork();

  if (pid == 0) {
    // child
    close(in_pipe[1]); // close writeend
    dup2(in_pipe[0], STDIN_FILENO); // replace stdin with read end of pipe
    close(in_pipe[0]); // close read end since it has been duplicated

    // exec the program "./numbers" with no command line args
    string command("./numbers");
    char* args[] = {"./numbers", nullptr};
    execvp(command.c_str(), args);

    // should NEVER get here
    return EXIT_FAILURE;
  } else {
```

Exercise: fill in the blanks



```
} else {
  close(_in_pipe[0]_______________); // close read end

  // write inputs to the pipe
  string inputs = "30\n40\n50\n6";
  wrapped_write(to_echo, __in_pipe[1]______);

  // close pipe so that exec'd
  // program knows there is no more piped contents to read
  close(___in_pipe[1]_______);

  // wait for child to finish
  waitpid(pid, nullptr, 0_);
}
```

Exercise 3

Exercise 3: What does this print? Does it terminate?

```cpp
int main(int argc, char* argv[]) {
  array<int, 2> pipe_fds {-1, -1};
  pipe(pipe_fds.data());
  pid_t pid = fork();
  if (pid == 0) {
     dup2(pipe_fds.at(0), STDIN_FILENO);
     close(pipe_fds.at(0));
     // cat should read from stdin till eof, printing everything it reads
     vector<char*> args {"cat", nullptr};
     execvp(args.at(0), args.data());
  }
  write(pipe_fds.at(1), "the city in rain", strlen("the city in rain"));
  close(pipe_fds.at(1));
  close(pipe_fds.at(0));
  waitpid(pid, nullptr, 0);
}
```

Exercise 3: What does this print? Does it terminate?

```
int main(int argc, char* argv[]) {
  array<int, 2> pipe_fds {-1, -1};
  pipe(pipe_fds.data());
  pid_t pid =  fork();
  if (pid == 0) {
     dup2(pipe_fds.at(0), STDIN_FILENO);
     close(pipe_fds.at(0));
     // cat should read from stdin till eof, printing everything it reads
     vector<char*> args {"cat", nullptr};
     execvp(args.at(0), args.data());
  }
  write(pipe_fds.at(1), "the city in rain", strlen("the city in rain"));
  close(pipe_fds.at(1));
  close(pipe_fds.at(0));
  waitpid(pid, nullptr, 0);
}
```

Print: `the city in rain`

It doesn’t terminate since the child has its write end open, thus cat never reads EOF

Homework 4 Overview

Overview

- In HW4, you will be implementing a simplified shell

- This shell only needs to support variable length pipelines

- You can reuse the same docker container as the one setup from the Project, thus allowing you to use the boost functions.
  - Highly recommended, the string functions will make parsing user input a lot easier.

HW4 Provided Files

- We provide some files to get you started

- Sample C++ programs:

- **`sh.cpp`** gives an example of a program that uses execvp

- **`stdin_echo.cpp`** does the same thing as “cat”, it reads from stdin 1 line at a time, prints what it reads and repeats until EOF. But you can modify the code and run it as a command for your pipe_shell to help with debugging.

HW4 Tests

- We provide the test cases:
  - **`tests`**: a directory containing all of the tests
  - **`test_files`**: a directory containing files used for the tests

- To run a test:
  - **`./pipe_shell < tests/simple_input.txt > out.txt`**
    Runs your pipeshell giving it the input for the “simple” test case and writes the output of your pipe_shell to **`out.txt`**
  - **`diff out.txt tests/simple_output.txt`**
    Compares your program output (**`out.txt`**) to the expected output to the simple test case
    If nothing is printed, then there are no differences between the files and your code passes!
  - Replace “simple” with one of the other test cases in the **`tests`** directory to run that case.

HW4 Demo

Any Questions?