ICSI412 Project 1: Documentation

Huang Kaisheng (2020215138@stu.cqupt.edu.cn)

Apr. 30th, 2023

Contents

1 System Documentation

This section contains a high-level data flow diagram, list of routines and brief descriptions and some implementation details.

1.1 High-level Data Flow Diagram

Figure 1 shows the high level data flow diagram.

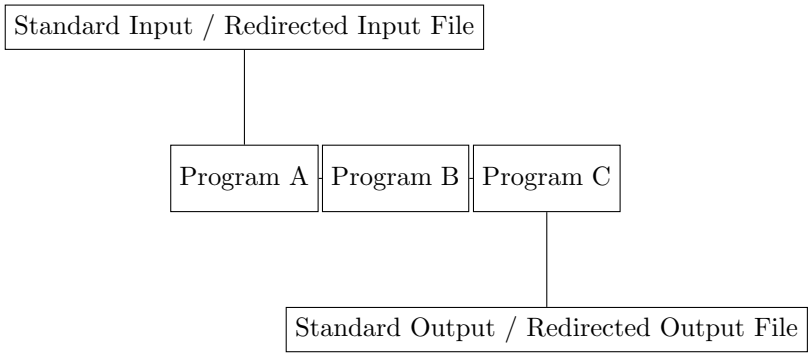

Figure 1: Data Flow Diagram

1.2 List of Routines and Brief Descriptions

Function	**Description**
`execute_command`	Forks and executes the commands specified in the `command` array. The `input_fd` parameter specifies the file descriptor to replace the standard input, and `output_fd` specifies the file descriptor to replace standard output. Returns -1 if an error occurs, and 0 otherwise.
`execute_piped_command`	Creates a pipeline between commands specified in the `commands` array and executes them. Uses the file descriptor array specified in the `pipes` parameter for communication between commands. The `file_remap` parameter specifies the stdin and stdout redirection map. Returns -1 if an error occurs, and 0 otherwise.
`parse_command`	Parses the command string specified in the `command` parameter, and stores the results in the `commands` parameter. The stdin and stdout redirection map is stored in the `file_remap` parameter, and the number of commands parsed is stored in the `num_commands` parameter. Returns -1 if an error occurs, and 0 otherwise.

1.3 Implementation Details

The program shown above is a shell program that allows the user to input commands and execute them.

The `execute.c` file contains two functions:
1. `execute_command()` takes a command array, input file descriptor and output file descriptor as arguments. It forks a new process to run the command and redirects stdin and stdout to the input and output file descriptors, respectively. It returns 0 if successful and -1 if there was an error.
2. `execute_piped_command()` takes a 2D array of commands, an array of file descriptors for pipes, an array of file names for input and output redirection, and the number of commands as arguments. It creates multiple pipes and forks a new process for each command. It redirects input and output either to the appropriate pipe or the specified input/output file. It returns 0 if successful and -1 if there was an error.

The `mysh.c` file is the main program that reads user input, parses the commands, and executes them. It uses `fgets()` to read input from the user, and passes the input to `parse_command()` and `execute_piped_command()` to parse and execute the commands, respectively.

The `parser.c` file contains the `parse_command()` function, which takes a command string, an array of command arrays, an array of filenames for input/output redirection, and an int pointer to the number of commands. It splits the command string into tokens and saves each token to the appropriate command array. It also handles input/output redirection and single quotes in the command string. It returns 0 if successful and 1 if there was an error.

2 Test Documentation

This section contains the description of test method, and a test result.

2.1 Test Method

I wrote a naive shell script to make the test, whose procedure is:
1. Read the test suite definition and make two temporary directories, one for our shell and another one for the system shell, to compare the result.
2. Run `make` to build the program.
3. Run our shell with the shell script in test suite, and run the system shell with the shell script in test suite.
4. Run `diff` to compare the results and check if the output is same.

2.2 Test Result

With the test suite provided in the homework document:

```
victor@victor-dev:~/CSI412/Project1(master)  make test
./test/test.sh countryCity
```

```
make[1]: Entering directory '/home/victor/CSI412/Project1'
gcc -c -o mysh.o mysh.c -I.
gcc -c -o execute.o execute.c -I.
gcc -c -o parser.o parser.c -I.
gcc -o mysh mysh.o execute.o parser.o -I.
make[1]: Leaving directory '/home/victor/CSI412/Project1'
Using test suite countryCity:
cat country.txt city.txt  | egrep 'g' | sort | more > countryCitygSorted.txt
cat country.txt city.txt  | egrep 'g' | sort | wc -l > countryCitygCount.txt
Testing program:

Making system shell sample:
+ cat country.txt city.txt
+ egrep g
+ sort
+ more
+ cat country.txt city.txt
+ egrep g
+ wc -l
+ sort
Making diff...
 countryCitygSorted.txt       TEST PASSED!
 countryCitygCount.txt        TEST PASSED!
Removing artifacts...
```

Running screenshot:

Figure 2: The screenshot of shell

3 User Documentation

This section contains how to build the program and argument definition.

3.1 How to run the program

Make sure that there is a C compiler and GNU make in your environment.
Run `make` to compile the program.
And the binary will appear at `mysh`.

3.2 Argument Definition

The program receives 1 optional argument, which is the script to be interpreted.