Joe Uzdzinski
Project 2: Shell Documentation

This project is a basic implementation of a linux shell.

The core of this shell is a loop that runs until an exit command is issued. A generalization can be seen in this pseudocode:

```
while (true)
        if (batchfile present)
                get next command from batchfile
        else
                get command from user (keyboard)

        compartmentalize command into subparts (command, file, special operations, etc.)

        if (pipe present)
                send command to pipeHandler()
        else
                if (internal command)
                        send to internalCommandHandler()
                else (external command)
                        send to externalCommandHandler()
```

The program is split into a few different modules: main.c, parser.c, internal.c, and external.c.

main.c

The main function holds the main loop which runs the program and its structure is shown above in the pseudocode. This file contains structure, but for its' functionality it calls functions that lie in the other modules.

parser.c

This file holds the functionality for the parsing functions, data manipulation, and file descriptor and pipe handling functions.

getParsedCommand( ) and getBatchCommand( )
These two functions have primarily the same purpose, to get the next command to run in the shell. The first will ask for a command via a prompt ("MyShell $") and the latter will already have had the command passed into it from main which grabbed the next command from the batch file.

splitInputHandler( )
This function will split the next command in the loop into parts based on if there are any special operations present. For example, if it recognizes a dual redirect (input and output redirect) in the command it will separate the command, input file, and output file into a struct called splitInput which has variables that will hold each one. This provides a more simple way to execute the command and set up file descriptors in later functions.

fdHandler( )
This function is called right before any command is ran. If any special operations are detected in the command, then the proper file descriptors will be changed. After the command is executed, the descriptors are closed (not in fdHandler( )).

pipeHandler( )
This function will handle any command that contains a pipe. This is the first thing main will check for after getting a command as you can see above in the pseudocode. A fork is done in the function. The first (input) command is executed in the child process and the second (output) command is done in the parent process. The two processes are able to communicate with each other via the usage of a pipe.

nullifyArray( )
This function takes a pointer to an array of char pointers. It then sets each value in the array to NULL so the array can be used in the next iteration in the loop without any problems. This function is called at the end of the while loop.

printArray( )
This function just prints the contents of an array of char pointers. This function was used for debugging and testing purposes.

internal.c

This file has two functions that deal with the internal commands of the program.

checkInternal( )
After a command is parsed and split, it will be checked to see if its an internal command by this function. It simply includes one if statement that compares the command to the list of internal commands. If any match, the function will return 1, 0 if otherwise.

internalCommandHandler( )
Every internal command is executed via this function. A series of if else if statements check if the command is equal to the corresponding internal command. Once it is found, the internal command will be executed by the code after the if statement.

external.c

This file only contains one function, the external command handler

externalCommandHandler( )
This function handles all commands that are not implemented internally via program invocation. A fork is performed and the command is run via an exec statement in the child process. If needed, the fdHandler( ) sets up the proper file descriptors prior to executing.

structs.h

This file defines four structs used throughout the program.

ENV
The environment struct holds the current working directory while navigating a terminal and the shell directory that the shell executable is sitting in.

CMDINFO
The command info struct holds information about each command, particularly about the special operations. It has an int for each special operation. They are initialized to zero, but if a special operation is detected, its corresponding variable will be set equal to 1.

INPUT
The split input struct holds a carrier for a command, a second command (if pipe is present), an input file, and an output file. This makes processing commands and setting up file descriptors easier in the handling functions.

FILED
The file descriptor struct is used to save variables that pertain to the redirecting process, so different parts of the process can happen in different functions without having to pass so many individual variables.

testing

The shell was tested thoroughly after each functionality was implemented. The parser was written first, internal commands next, and then lastly the ability to execute external commands. A simple printArray function that I wrote did a lot of the heavy lifting here. It was used to make sure the proper output was stored in the argsPtr and the carriers inside the INPUT struct. The CMDINFO struct also made it easy to know exactly where to send each command during its iteration of the shell loop. Modularity was also something that was kept in mind during the process. Most processes, if not belonging to their own source file, lied in their own function.

```c
//prints each element
void printArray(char** arrPtr) {
    int j = 0;
    while (arrPtr[j] != NULL) {
        printf("%s ", arrPtr[j]);
        j++;
    }
    printf("\n");
}
```

```c
typedef struct _COMMAND_INFO {
    int amp;
    int inputRedirect;
    int outputTruncate;
    int outputAppend;
    int dualRedirect;
    int pipe;
} CMDINFO;

typedef struct _SPLIT_INPUTS {
    char * command[100];
    char * command2[100];
    char * inputFile[100];
    char * outputFile[100];
} INPUT;
```