Name: Georgios Karagiannis

Design Document

The data structure used to create the shell is a singly Linked List. Every Node in the Linked List represents a command. So, for example if we have 3 commands in the form command1 | command2 | command 3, then we would have three nodes in the List. The first node in the list is the first command and thus we insert at the end of the List (at the tail). If a node A points to another node B, then A's output will be B's input. Also to execute B, we will have to fork off A and thus B will be the child process of A. This is why I store the different commands in the linked list, so that I know which command precedes the other.

Every Node has the following fields:

1. a command string
2. an input string that holds the name of the input file
3. an output string that holds the name of the output file
4. a string array (**char) that holds the flags of the command (if any)
5. an integer that hold the number of flags
6. a node reference that points to the next node in the list (can be NULL, if there is no other node)

The Linked List contains a head and tail Node references that point to the first and last element of the List respectively. Also it contains an integer that holds the number of elements in the list.

Flow of the program

The flow of the program goes like this:

1. the args array, that holds the command line arguments, is populated by calling getline(), which is given to us from the lex file.
2. Check if the user typed "exit" or "cd" and call exit(0) or chdir() accordingly.

3. A linked list that will hold the command(s) is created through the constructor (described below).
4. Initialize the linked list by parsing the args array.
5. call the recursive execute() function.

To initialize the linked list, we have to first parse the args array and figure out how many instructions there are, what their input, output files are and if they contain any flags. The function that does that is called init().

init() takes 6 arguments and returns void:
1. a command string
2. an input file
3. an output file
4. a string array for the flags
5. a linked list

When we first call init() (from main) we give it NULL for all the arguments except the Linked List, which will be the data structure that will hold all the commands. The purpose of this function is to populate the first four arguments and put them in a node in the Linked List. All the arguments are passed to the function by reference. Every node in the List has a command, an input file (may be NULL), an output file (may be NULL), a flags array (may be NULL), and the number of flags (may be 0).

To populate the Linked List, we do the following:

We know that the first argument of the args array is a command. Thus, we populate the command argument with args[0]. Then we want to traverse args to see if the first command has flags and/or input, output files. Thus we have a for loop that goes through args from element 1 (not 0, as element 0 is a command) till the end.

In the for loop, we check if the current element is a special character ("<", ">", "|"). If the previous element is a "|", this means that the current element is a command. This makes sense, since the only thing that can follow a pipe is a command. Note here that we can access the previous element since we start from element 1 of args and not 0.

If the current element is either a “<” or a “>” then the next element is an input or an output file respectively. Thus we can populate the second and third arguments (input, output files).

if the current element is neither a command nor an IO file and there it is not “<”, ”>”, ”|” characters then it is a flag.

Now we have populated all the arguments (provided that they exist in args).

If a pipe occurs, then we have to insert everything that we have so far (command, input/output, flags, number of flags) to the linked list, so that we can get the next command following the pipe and insert it in another node. So, we call the insert function described below and clear all the variables so that they can get repopulated.

If there is no pipe and the current command is the last one, then we just insert to the List after the loop has executed.

After the init function has executed, we have a linked list that contains nodes that contain the simple command (e.g “ls”, “grep”), the input file, the output file, the flags (e.g -la, -v) and the number of flags.

Now that everything is set up, we need to execute the commands in the linked list. **The recursive function execute() is called.**

execute() takes as **arguments** a Node and an int array used for piping. We pass a node, because at every recursive call a command in the specified node is executed. We start from the head of the List from main() and at every recursive step we call the next node that the current node is pointing to (if it’s not NULL). This enables infinite piping and is in sync with the data structure that holds the commands, as I said in the first paragraph.

Every time we want to call the function on the next node, we fork off the current node. **This means that we create child processes to execute each pipe instruction. 3 instructions would mean 3 processes.**

Initially we call execute() from main with the head of the list and a bogus int array for piping. The array is bogus (initialized as {-1,-1}), so that we know that we call it for the first time, where there is no piping. Every time we recursively call the function, we get the input of the instruction we want to execute, from the output of the previous one.

execvp() takes 2 arguments. The first one is a string of the simple command (e.g. ls, grep) and the second is a NULL terminated string array that contains both the simple command and its flags, if any (e.g. ls -la, grep -v “word”). So, I have created another function called **get_whole_command()**, which takes an empty string array and a Node reference and it **merges the simple command with its flags and puts it in the first argument**. So for every node, we call get_whole_command() to get a string array which has both the simple command and its flags. For example, if the simple command was “ls” and its flags were “-l -a” now get_whole_command() gives “ls -l -a”, in a NULL terminated array of strings.

Once we do that, we need to check if the current node is part of a pipe or is the first command. This is easy to check. If the int array has bogus values, then the current node is the first one in the list**. Else, we need to redirect the input from STDIN to p[0], where p is the pipe array**. Now, we get the input from the output of the previous node.

After that we check if the current node has input, output files. If yes, we redirect input/output. Else we use STDIN/STDOUT.

If the current node points to another node, then we need to make a recursive call passing as parameters the next node in the list and a pipe array. We fork and make the recursive call in the child and in the parent we redirect the output from STDOUT to the pipe (p[1]). **We do that because the next node needs to get its input from p[1] and not STDOUT**. In general p[0] is used for input and p[1] for output. After the forking we execute the current command with execvp.

If the current node doesn't point to another node, we don't do any piping; we just execute with execvp.

All the above is of most importance regarding the shell program. Here's a description of functions that are not described above:

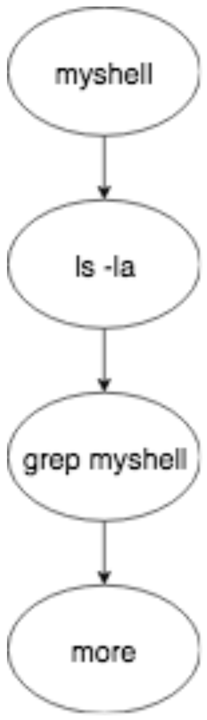
1. newNode(): the constructor of the node object
    a. Arguments:
        - command character
        - input file string
        - output file string
        - flags string array
        - Number of flags integer
    b. Returns a node object
2. newLinkedList(): the constructor of the Linked List structure
    a. Arguments: None
    b. Ruturns a Linked List Structure
3. freeNode(): destructor of the Node structure
    a. Arguments: a pointer to a Node
    b. Returns void
4. freeLinkedList(): destructor of the Linked List structure
    a. Arguments: a pointer to a Linked List
    b. Returns void
5. insert(): Inserts a node at the tail of the List
    a. Arguments:
        - command character
        - input file string
        - output file string
        - flags string array
        - Number of flags integer
        - A Linked List
    b. Returns void

6. error_report(): uses perror() system call to output any errors, if the file descriptor argument is less than 0.
    a. Arguments:
        - A file descriptor integer
        - A string where the error originated from
    b. Returns void

Notes

1. When an error occurs in the execute() function, we kill the current process. For example, if the input file specified cannot be opened because the name is incorrect, then we kill the process that is trying to open it. This happens because the only way for a process to return to its parent is by execvp(). In execute(), execvp() is called last, **so if the process doesn't get to execvp(), it will exist forever. Thus we need a way to kill the process and return prematurely to the parent.** A process gets killed by using the kill() system call.
2. In main(), we fork one time and the child process calls execute. If we assume that the parent process is myshell, then its child will execute the first command. If needed, the child will fork as many times as the number of pipes in the command. Thus myshell waits for all the subsequent children to finish (either by reaching execvp() or by getting killed).

   For example, if we had the command ls –l | grep myshell.c | more here is what would happen graphically:


myshell
ls -la
grep myshell
more