mysh

A pipeline of commands

The `exec()` method in `exec.c` takes in a `Pipeline` structure, which consists of a linked list of Commands each having their own linked list of arguments. Each command would be piped together to be executed simultaneously. First, we must initialize our file descriptors: one for `STDIN` and another for `STDOUT`. Now, we iterate through each, but not the last, command in the pipeline. For each iteration, we use the `pipe()` function to allow for the reading and writing of data between commands.

We run our command in the child process. Send our original file descriptor (`in`), the file descriptor of the write end of the pipe (`fd[1]`), and our current command to the `runPipe()` method. Here, we check if each file descriptor matches their original value, and if not, we duplicate them over standard input/output and close them afterwards. Like this, there would not be a bad file descriptor when data is being passed in our pipe. After execution, we close the write end of the pipe as it is no longer needed. We still need the read end of the pipe kept open for the next child process.

On the next iteration, the output of our previous command is sent as the input for our next command. Repeat the above steps until there is only one command left in the pipeline. Send our command, along with its file descriptors, to the `runPipe()` method one last time. On this step a new pipe is no longer needed, so we close the read end of the pipe afterwards.

Commands with or without arguments

The first node of the linked list `Command` is chosen as our current command. This would be run in the `runPipe()` method by calling the `execvp()` function in a child process. `execvp()` accepts two parameters: the first being the file you wish to execute (i.e. our current command), and the second being an array of null-terminated strings (the list of arguments a command may have).

So, we need to convert our command from a linked list to an array. Pass our current command, along with an array of zeros, by reference to the `cmdArgs()` method. Set the first element of the array as the name of the current command. Next, we check if our command has any arguments. If so, we iterate through each argument and add it to the array. If not, then the array would only contain the name of the command. We now have a command for `execvp()` to execute.

Commands with input/output redirection

Before our command is executed, however, we need to check if the command contains angle brackets (< or >). Regardless of symbol, we check if a file exists and open it to capture its file descriptor. Then, we duplicate the file descriptor over standard input/output and close them afterwards. Like this, the child process has `STDIN` coming from an input file, `STDOUT` coming from an output file, and no extra files are left open. It is now safe for the command to be executed.