

## Pseudocode for Linux Shell program.

```
#define
Int 0 = internalCommand
Int 1 = externalCommand
#define
Int 2 = PipeCommand
Int 3 = redirectiCommand

// This is the pseudocode for the main function which will contain the main while loop
// this is to demonstrate the order which I think the logic should be handled
// details for the algorithms I intend to use for each function can be found below the main function's pseudo code.
```

**Main (argc, \*argv) {**

**Boolean flags**

```
Piped = 0;
Redirect =0;
Batch =0;
// flag to tell if we should execute immediately this will be set to 1 if we detect & at the end of cmd
Int execNow =0;
```

First check if main was provided any args.  
If so then assume that the arg was a file and close stdin  
Then attempt to open the given file and run commands from that as stdin

```
// display welcome message and great user
```

```
welcomeMsg ();
```

```
    // run program until user selects to exit
```

```
    // main while loop
```

```
    While (running == true)
```

```
    {
```

```
        // display prompt which should show working directory
```

```
        promptUser ();
```

```
        Str = getUserInput ();
```

```
        // parse input
```

```
        parseArgs(Str);
```

```
        // if command is internal command
```

```
        If (internal command)
```

```
        {
```

```
            HandleInternal()
```

```
        }
```

```
        else if(!internal command)
```

```
        // else assume this is a file in current directory to open or a program from the path variable
```

```
        {
```

```
            HandleExternal() ;
```

```
    }  
    Else print stderr  
    // if all else fails the display error message  
  
} // end while  
  
free memory if needed  
  
} // end main
```

```
Int parseArgs(){
```

```
strokc(input," ") = token;
// while the string is not null break it apart using white space as delimiter
//also we will push each separate command to the arguments array
while (strokc(input," ") != NULL) { argc++; argv[i] = token; i++;}
// Store the number of strings in the command in the integer variable argc
//Store the C-Strings in an array of character pointers
```

```

Look at each arg to search and see if there is a pipe command
If(!arg(i) == | ; i++)
If we get to the end of the list then we know there is no pipe command
else
Piped = 1

```

```

Look at each arg to search and see if there is a & command
If(!arg(i) == &; i++)
If we get to the end of the list then we know there is no & command
else
execNow = 1

```

```
welcomeMsg () {
```

```
}; // end welcomeMsg
```

**getUserInput ()**{

*//getUserInput will make one long string of chars containing everything typed*  
use readline to read in user input until user hits enter  
stores input in a c string variable

}// end getUserInput

**promptUser ()**{ print the name of the working directory followed by name if shell};

*// Enter switch case to handle internal command*

*//return 0 if successful*

**HandleInternal()**{

Switch (internalCommandType)

*// quit*

Case{

*Running = false;*

*//say bye to user*

*Printf(that's all folks!);*

*Return 0;*

} break

*// cd*

Case{

*//change working directory to input argv[0]*

*Use the chdir() function and pass it the 1 index in the array of arguments*

*•If the <directory> argument is not present, print the current directory*

*•If the specified directory is invalid, generate an error*

*•This command should also set the PWD environment variable for the shell to <directory>.*

} break

*// dir*

Case{

*Use dirent function to view current directory*

*while ((newdirectoryEntry = readdir(newDir)) != NULL)*

*print out each item in directory*

} break

*// clr*

Case{

*// this clears the console using escape sequences*

*// printf("\033[H\033[J")*

} break

*// echo*

Case{

*// just reprint argv[0...n]*

} break

*// help*

Case{

*// print the user manual using printf()*

} break

*// pause*

Case{

*// don't do anything until user hits enter again*

} break

*// environ*

Case{

*// print the environment strings*

*•Current user*

- User's home path
- Shell name
- OS type
- Hostname
- Directories to search to find an executable.

```
} break
```

```
} // end handleInternal
```

//this function should be able to handle all external commands that exist, both piped and redirected.

**HandleExternal() {**

// here we will assume that the argv[0] is the first argument in an external command

// we fork then exec the child of the fork giving it argv[0] as the first command, and up to argv[argc]

    If pipe = 1

*Call handle pipe to exec*

    HandlePipe();

    else if redirect = 1

*Call handle redirect to exec*

    HandleRedirect();

    else

*//just exec normally*

    execArg(char\* cmd , char\*\* args)

```
} // end HandleExternal
```

// handle pipe should be able to pipe N amount of commands along with their args together

**HandlePipe(cmd\*[] list of commands, args\*[] list of args) {**

Create a pipe before we fork

Close std in

Close std out

Make array to assign new pipe as std in and out

Pipe[2]

Iterate through list of commands

For (int i =0 ; i < listofcommands.len; i++)

{

*//if we are the new process*

    if (fork() == 0)

    {

*// by using modulo we alternate between reading and writing for n amount of pipes*

        Open std in Write to pipe[ i % 2]

*// exec the i'th cmd with its associated args*

        Exec(cmd[i], argv)

    } else {fork failed}

} // end for

```
}; // end handle pipe
```

// handle redirect should identify what kind of redirection this is(i.e. ,redirecting input/output)

// redirect N amount of commands and their associated args together

**HandleRedirect(cmd\*[] list of commands, args\*[] list of args) {**

Create a pipe before we fork  
Close std in  
Close std out  
Make array to assign new pipe as std in and out  
Pipe[2]

Iterate through list of commands

```
For (int i =0 ; i < listofcommands.len; i++)  
{
```

**Check to see what type of redirection this is(concat vs truncate/ in /out)**

**If redirecting input then open new input stream with pipe as file**

```
int newstdin = open(pipe[0]O_RDONLY);
```

**else if (appending out put) // use pipe output end as file to take output from**

```
int newstdout = open(pipe[1]O_WRONLY|O_CREAT,S_IRWXU|S_IRWXG|S_IRWXO);
```

**else if truncating // use pipe output end as file to take output from**

```
open(pipe[1]O_WRONLY|O_CREAT,S_TRUNC,S_IRWXU|S_IRWXG|S_IRWXO);
```

```
//if we are the new process
```

```
if (fork() == 0)
```

```
{ //its output or input should be inherited from parent
```

```
// exec the i'th cmd with its associated args
```

```
Exec(cmd[i], argv)
```

```
} else {fork failed}
```

```
} // end for
```

```
}// end HandleRedirect
```

*//this will be the function that gets called to exec any desired external program*

*// it takes the command to be executed along with any arguments to be passed with it*

**execArg(char\* cmd , char\*\* args)**

```
{
```

```
int pid = fork();
```

```
// if 0 then exec because we are in new child process
```

```
if fork == 0 {
```

```
exec(arg[0], argv);
```

```
}
```

```
// if not 0 then we are in main process ,i.e the shell, and we should check if we need to wait or not
```

```
// for child to compete
```

```
else {
```

```
if (execNow ==0)
```

```
wait(pid)
```

```
//other wise we don't wait because we detected &
```

```
} // end else
```

```
} // end execArgs
```

```
////////////////////////////////////  
////////////////////////////////////TESTING_PLAN////////////////////////////////////  
////////////////////////////////////
```

My plan to test the shell is to create various batch files using the piping and redirection in different combinations with different commands and arguments, and to check if they work as expected.

For testing purposes i also will be using hardcoded values of arguments to check that my parser is working as expected.

The part that I will need to test the most is probably the piping and redirection. This will be accomplished by executing identical commands on an actual Linux shell and comparing results.

Any unexpected errors in logic, or code will be handled with tried and true debugging methods such as selectively commenting out portions of code and inserting print statements strategically. This used in combination with the gdb debugger will allow me to trace my algorithms step by step and fix any problems as they arise.