Eggshell

an Operating Systems project

CPS1012

Andre' Jenkins

76999M

Contents

Chapter 1

Structure of the code

The code was structured into different directories, so that they may be organised according to what they are supposed to achieve.

1.1 Main files

The files here are ones which are used in the front layer of the eggshell. Here you may find a main C file that uses the eggshell functions, the main eggshell `.c / .h` files which either call multiple other functions, or are simple enough to be on the front layer, and other files such as the Makefile, any scripts, testfiles, and other miscellaneous files.

main.c — The main `C` file that the executable is retrieved from. Uses libraries such as `eggshell` and `linenoise`.

eggshell.c/h — The `eggshell library` used by the main file in order to start the eggshell and utilise it. Uses multiple libraries that are all found in the `src` directory.

Makefile/Makefile-GCC — The Makefile necessary to generate the executable. The current default Makefile uses the `Clang` compiler, for reasons stated in the `README.md` file. To use the Makefile that utilises the `GCC` compiler instead, either change the name of the `Makefile-GCC` file, or run `switch.sh`

switch.sh — A script that aids in switching compilers for the makefile. This was written for each switching between `Clang` and `GCC`, due to the ease of debugging with `Clang`, and the standard nature of the `GCC` compiler.

README.md — The `README` file holds the instructions to compiling the program, as well as a quick summary of the program and its utilities.

testinput.txt — A file used to test the capabilities of the eggshell. Running `./eggshell test` will immediately launch the eggshell and run this script, to provide a quick testing method.

LICENSE, .gitignore & .yml files — Files that are unimportant to the project itself. These were used for `git` purposes, as the project was also uploaded as a `git` repository.

1.2 Source files

The files found here are the bulk of the code making up the eggshell. In here, every single `eggshell header library` is present, all of them with their own specific and complicated purpose. These were seperated from the main `eggshell` file in order to organise the core of the project from the specific elements making up the project itself.

variables.c/h — Contains all the functions and structs relating to the variables created and stored by the `eggshell`. For example, all the **shell** variables can be found here.

printer.c/h — The main file dealing with the `print` command.

proc_manager.c/h — The file dealing with the execution of external commands.

sig_handler.c/h — Contains the `signal handler` function used in order to suspend and interrupt processes. Also contains an additional function in order to reawaken a suspended process.

redirection.c/h — The main file dealing with `input/output` redirection.

pipe_manager.c/h — Contains functions dealing with the piping system that the `eggshell` offers. Also contains a special execution function, rather than using the one found in `proc_manager.c/h`

1.3 Other files

There are also other directories that contain files that aren't integral to the functionality of the eggshell, but are still related somewhat.

documentation/ — Contains the `.tex` file that generated this report, as well as other items related to it. In order to recompile it, you'd most likely need to install TeXlive first.

ci/ — Unrelated to the main project. The Makefile here is used for Continuous Integration for the `git` repository.

add-on/ — Contains the `linenoise.c/h` library that was used in the main file to simulate a terminal's prompt with input.

.vscode/ — Contains files that helped with debugging/building the project in `Visual Studio Code`.

Chapter 2

Code documentation

2.1 `main.c`

2.1.1 Use of `main.c`

This code is mainly used in order to produce an executable using the `eggshell.` This is because the eggshell is mainly used as a sort of `API,` which interfaces with the inner functions that the eggshell offers.

As a result, the `main` implements some not-so-integral elements of the eggshell, such as an introduction/boot-up screen, an additional `test` argument feature, and some `linenoise` functions such as history *(ability to use ↑ and ↓ keys to traverse through older commands)* . It also uses the function `updatePrompt();` which updates the prompt of the eggshell in order to display the current directory.

A thing to note is that the external command `clear` is run by the line:

```
runLine("clear", "");
```

...which is an external function found in `eggshell.c` .

Note: In order to test the first part of the program, which loads `testinput.txt` and runs it, you'd need to run the eggshell with the `test` parameter: `./eggshell test`

2.2 eggshell.c

2.2.1 Use of eggshell.c

This file serves as the **core** of the eggshell itself. Its main method, `execute` , executes the line inputted by the user. Almost every other function of the eggshell is accessed through this main file automatically, using parsing.

2.2.2 void initEggshell ()

All this function does is initialise the eggshell. To initialise, all it needs to do is initialise its shell variables, which is does by calling the method `initShellVars()` from `variables.c`

2.2.3 void runLine (char *command, char *line)

This function executes the command `command` with the arguments found in `line` . An example of this is:

`command` = `"chdir"` and `line` = `"../src/"`

Which would execute `chdir` with arguments `../src/` . It does this by having a significant if/else-if block that checks which function to execute:

```
if(strcmp(command, "print") == 0) printLine(line);
else if(strcmp(command, "all") == 0) showShellVars();
else if(strcmp(command, "vars") == 0) displayUserVars();
else if(strcmp(command, "chdir") == 0) changeDirectory(line);
else if(strcmp(command, "source") == 0) runScript(line);
else if(strcmp(command, "fg") == 0) resumeProcessSignal(FOREGROUND);
else if(strcmp(command, "bg") == 0) resumeProcessSignal(BACKGROUND);
else externalCommand(command, line);
```

The signal handler is also initialised by calling the `init_handler()` function in `sig_handler.c` .

2.2.4 void changeDirectory (char* directory)

This function changes the directory to the directory specified by the parameter. This is done using the inbuilt `chdir` function.

If changing the directory was successful, the `$CWD` shell variable is updated, and the new directory is displayed.

However, if it was failing, `perror` is used to display the error message.

2.2.5 void runScript (char *filename)

All this function does, is load a `script` file which can have any extension, and execute its commands line by line. It contains measures, such as ignoring any line starting with `#`, or any empty lines. This effectively enables commenting support for scripts using the `#` symbol.

It also emulates a prompt, so that the output generated when running the script could be more readable to the user.

2.2.6 `void execute (char *line)`

This function is the main one to be executed. It is what reveals the eggshell functions to the user, as all other functions are accessed via this one.

What it does first of all, is check the line's structure in order to parse it for specific cases. For example, in this section here:

```
27 if(strcmp(line, "exit") == 0){runLine("clear", ""); exit(0);}
28
29 // Check for variable assignment
30 if(parse_var(line) == 0){return;}
31
32 if(pipe_parser(line) == 0){return;}
```

The first line checks whether the line is identical to the exit command `exit` , at which point the `clear` command is run, and the program stops executing. Line 30 and 32, use functions from `variables.c` and `pipe_manager.c` respecively, to check whether the line assigns a variable, or contains pipes in it. An exitcode of 0 from these functions means that they were detected.

If none of these are detected, the function then attempts to check whether the line contains any redirection symbols such as `>` , `>>` , `<` or `<<<` :

```
// Checks for different output redirection symbols
char* redirect_to_file  = strstr(line, ">");
char* append_to_file    = strstr(line, ">>");
char* input_from_file   = strstr(line, "<");
char* input_here_string = strstr(line, "<<<");

int out = 0, in = 0; // flag variables for redirection
```

The results of the `strstr` function used are stored in variables. These variables have two uses: They are used as flags to check which redirection symbol was found, **and** they point to the first character of the symbol itself, which is useful. The flag variables `out` and `in` , also serve a purpose which will be gone into later on.

After this, a conditional `if-else` block is used to parse the line for redirection. Depending on which `char*` flags where set, the respective function from `redirection.c` is called:

```c
// Checks for redirection
if(append_to_file != 0 || redirect_to_file != 0){
  filename = out_redirect_parse(append_to_file, redirect_to_file, line);
  out = 1;
}
else if(input_here_string != 0 || input_from_file != 0){
  filename = in_redirect_parse(input_from_file, input_here_string, line);
  in = 1;
}
```

One can also note, that one of the integer flag variables `in/out` is also set to 1. This is used to call the appropriate redirection function later on.

After doing so, two lines are used. One uses the `strsep` function to seperate the command from the arguments, and the other calls the redirect initialisation function from `redirection.c`

```c
// Seperates the command from the arguments
char *command = strsep(&line, delimiter);

int filefd = init_redirect(filename);
```

After this, the actual execution of the function is done. Depending on the integer flags triggered, the appropriate block of code in the conditional loop is run. One begins redirecting the input, one redirects the output, and in the case that neither of these flags are set, the line is simply executed.

```c
if(out == 1){
  redirect_out(filefd);
  runLine(command, line);
  close_redirects(OUT, filefd);
}
else if(in == 1){
  redirect_in(filefd);
  runLine(command, line);
  close_redirects(IN, filefd);
}
else{
  runLine(command, line);
}
```

2.3 `printer.c`

2.3.1 Use of `printer.c`

This file handles the `print` command. It's main purpose is to print the line, being able to detect whether a part of the string is escaped in quotation marks, and replacing unescaped variable references with their value.

2.3.2 `void printLine (char *line)`

The main function of the file, it is the primary function to be called from `printer.c`. It achieves the aforementioned goal by splitting the line accordingly, and passing the segments of the split line to the other function in the same file.

The splitting is done using `strsep`, and an integer variable `escaped` was used to determine whether the segment was escaped.

For example, a line such as:

hello $USER, and wel"come to the $HOME"

Will be split into several pieces like:

hello — $USER, — and — wel — come — to — the — $HOME

...and the segments are then sequentially handed off to the other function. Note that after the `wel` segment, the `escaped` variable is flipped.

2.3.3 `void printSeg`
`(char *segment, int escaped)`

This function utilises its parameters to print the segment, replacing the segment [or a part of it] with a value if it is found to be a variable name.

If the `escape` variable is set to 1, then the segment is simply printed without considering the rest of the function. However, if it isn't, then the following steps are covered:

- **IF** the first character is '$', start reading what's after it.
  - **IF** the string contains any invalid variable characters *(any character that isn't a capital english letter)*, terminate reading, and consider what has been read to be the variable name.
  - Retrieve the value of the variable with the name that was read.
  - **IF** the variable exists, and the reading was stopped, print the value, followed by the rest of the segment.
  - **ELSE IF** the variable exists, and no invalid characters were found, print the value.
  - **ELSE**, print the segment.
- **ELSE**, print the segment.

For example in the line above, $USER would be replaced by the value, whereas $HOME would be printed as it is, as it was escaped.

A limitation of this is that if a segment contains $ anywhere that isn't its first character, it is ignored. For example, if $A is B, `$Aees` would be printed as `Bees`, whereas `a$Ae Lincoln` would remain the same, instead of being printed as `aBe Lincoln`.

2.4 sig_handler.c

2.4.1 Use of sig_handler.c

This file contains functions relating to any signals, or any effects resulting from signals. To be more precise, it contains a *signal handler* , and a way to wake up a suspended process.

This was split into a seperate file for sake of organization, as having it bundled with the `eggshell.c` file would have added more complexity to the already significant file.

2.4.2 void signal_handler (int signo)

This is the signal handler that is initialised by `sigaction` .

```c
void signal_handler(int signo){
    pid_t process = currentpid();

    // Catches interrupt signal
    if(signo == SIGINT){
        int success = kill(process, SIGINT);
    }

    // Catches suspend signal
    else if(signo == SIGTSTP){
        int success = kill(process, SIGTSTP);
        resuspended = 1;
        suspended_process[last_suspended+1] = process;
        last_suspended++;
    }
```

It is lacking in `printf` statements, as after some research, it was found that it is not `re-entrant` , or `async-signal-safe` , meaning that if, for example, in the middle of running `printf` in a signal handler, another signal is caught, the program might end up in a deadlock.

A `SIGINT` signal is handled by simply sending the signal to the current process, whereas a `SIGTSTP` is handled by sending the signal, then setting the `resuspended` flag to 1, after which the pid of the now suspended process is appended to an array of pid's using a variable that keeps count of how many suspended processes there are. This ensures that multiple processes can be suspended and restored normally.

2.4.3 `void resumeProcessSignal` `(int state)`

This function activates in the case of either the `fg` or `bg` commands. It's purpose is to resume a suspended process by calling a function from `proc_manager.c` . This is done in order to abstract the process managing *(with `waitpid` etc...)* from the signal handler, while at the same time allowing access to the pid of the suspended process.

The correct pid is handed off to the function by using the `last_suspended` variable. This makes process suspension seem like a stack, **last suspended, first resumed** . If the suspension succeeds, the `last_suspended` variable is decremented, and the exitcode is set to 0. However, if for some reason it does not succeed, there are two cases: *the process is unresponsive* or *the process was resuspended.* In this case, the exit code is set to reflect which outcome happened, `-1` being an error and `18` being an indication of another suspension *(18 is the SIGNO of SIGTSTP)* .

2.4.4 `void init_handler` `()`

All this function does is initialise the signal handlers that are to be used. It does this by use of `sigaction` rather than `signal` , for multiple reasons:

- `signal` performs differently on different systems, making it an unpredictable function to use.
- `signal` automatically resets the signal action back to `SIG_DFL` after use. This means that during the time in which the signal is triggered and the handler is reinitialized, *the system is vulnerable to additional signals, which would not be handled* .
- `sigaction` permits the blocking of signals, which if needed in the future, would be impossible to implement with `signal`

Other than the initialisation of the `sigaction` struct, the handler is initialised to deal with `SIGINT` and `SIGTSTP` signals accordingly.

2.5 `redirection.c`

2.5.1 Use of `redirection.c`

The purpose of this file is to handle all **input** and **output** redirection. It parses a line for any redirection symbols, initialises the redirection using a filename *unless* `<<<` *is detected* , and also starts the redirection, depending on its own flags and how they were set.

2.5.2 Parsing functions

The functions `out_redirect_parse` and `in_redirect_parse` both have very similar code that functions differently in other to be more specialised while also simplifying the code from having too many conditional blocks.

Both of them require three strings, **two** specifying pointers to the symbols, and **one** which contains the line itself.

These are passed in the parameter in order for the functions to know which action is supposed to happen, and how the line is meant to be parsed. This is because, in the case of `<` and `>` , one can simply split according to those symbols and be left with the two elements required, albeit with additional whitespace, whereas in the case of `<<<` and `>>` , one would need to perform some pointer arithmetic to split the strings accordingly.

Let's take the `in_redirect_parse` function as an example:

```c
char *filename;

if(in_string != 0){
  filename = in_string+3;
  ins = 1;
}
else{
  filename = in_file+1;
  inf = 1;
}
```

Here, `in_string` and `in_file` serve as both pointers to the symbol, and flags as to which symbol is present, with `in_string` pointing to `<<<` and `in_file` pointing to `<` . If the former was detected, `filename` is set to a pointer *three characters away* from the symbol, whereas if the latter was detected instead, `filename` is set to a pointer *one character away*. In both cases, the respective flag is also set.

This ends up setting the filename to whatever is after the symbol, as whichever symbol was pushed, the pointer was moved away from it anyways.

However, this leaves two problems - the left part of the line still hasn't been parsed yet, and the start of the filename may still contain whitespace.

These are both solved by the upcoming `strsep` and pointer arithmetic. The `line` string, which still points to the whole thing, is split according to the `<` delimiter *(`>` for output)*, causing the `line` string to now contain only the command. As for the filename, it is placed in a loop, where if its first character is a space, its pointer is incremented by one. This means that it does not matter how many whitespaces are behind the filename, they will all be removed by the loop. As for the command, any whitespace at the end will not interfere, as the `execution` function in `proc_manager.c` will ignore those.

2.5.3 `int init_redirect (char *filename)`

All this function does is open the correct file descriptor, depending on which flags were set. This is done using an integer variable `filefd`, which would hold the descriptor, and the `open` function. The differences between `>` and `>>>` is simply whether the `O_TRUNC` macro or the `O_APPEND` macro is used, and for `<`, it's as simple as using `O_RDONLY`.

However for `<<<`, the `filename` string would contain an actual string, rather than a filename. As a result, a temporary file `stdin.tmp` is created in order to store the string in, and the temporary file is then subsequently opened. This has the benefit of having the redirection be the same for both `<<<` and `<`, abstracting it from the actual redirection function.

2.5.4 Redirection functions

These are grouped together for the same reason the parsing functions are. The code is similar, however to avoid unnecessary long conditionals, they were split into two.

In this case, we'll take the example for the output redirection:

```
int redirect_out(int filefd){
    // Duplicates file transcriptor 'stdout' to saveout
    save_out = dup(fileno(stdout));

    // Sets stdout file transcriptor to our transcriptor 'filefd'
    if(dup2(filefd, fileno(stdout)) == -1){
      perror("Failed in redirecting stdout");
      setExitcode(255);
      return 255;
    }

    return 0;
}
```

The first `dup` call is used in order to duplicate the file descriptor of `stdout` to `save_out` . This is needed later in order to restore `stdout` .

Then, the `dup2` in the if-condition sets our readymade file descriptor, which was set in the `init_redirect` function, as the replacement for `stdout` . If this fails, the redirection aborts, and the error code is reported, as well as the exitcode being set accordingly.

However if it succeeds, 0 is returned, showing that the redirection occured successfully.

This differs from `stdin` in that a different variable is used to store the file descriptor for `stdin` , and `stdin` is replaced instead of `stdout` .

2.5.5 `void close_redirects (int direction, int filefd)`

What this function does is reset all of the redirection that was initialised and used so far. It does this by flushing `stdout` or `stdin`, depending on which kind of redirection occured, and then closing the file descriptor `filefd`.

Our saved file descriptor, `save_in/save_out`, then replaces the place our old, now closed file descriptor occupied, after which that is closed as well.

Depending on whether the temporary file was created, it is then removed, as we have no more use for it.

After doing so, to ensure that everything is reset, every single flag variable within the file is set to 0, including the `save_out/save_in` descriptors. This is so that the next time redirection happens, it would be as if a redirection never happened at all.

2.6 proc_manager.c