CPS1012 - Operating Systems and Systems Programming 1 - Project Report

Daniel Cauchi

Contents

Chapter 1

Introduction

This is the project report for CPS1012 Operating Systems and Systems Programming 1. During this project an eggshell was implemented which more or less mimics the Linux/Ubuntu bash. Multiple operating systems techniques were used alongside the C programming language in conjunction with the CLion IDE.

This report contains an overview of the program as it should be read if it were to be opened by a programmer who needs to understand how the system works. Smaller problems are tackled individually in order to build for the bigger problem, the bigger problem being the eggshell itself. In-line comments within the code were also made to further clarify every part of the program.

The code contains no known program-breaking bug, such as segmentation faults, since error checking is done throughout. Every part of the project was attempted and should work successfully as it was thoroughly tested, as is shown in chapter 3 of this project. Known issues and certain features which would have been nice to get added are also listed in chapter 4 of this report.

For each subsection in chapter 2, a brief explanation is given for anyone wishing to skim through the report and have an idea of how the program works. For a more detailed explanation of each function, a more detailed explanation is given.

Chapter 2

Implementation and Explanation

2.1 CMakeLists.txt

2.1.1 CMake file

```
cmake_minimum_required(VERSION 3.9)
project(AssignmentRedone C)

set(CMAKE_C_STANDARD 99)

add_executable(eggshell variableManager.c parser.c linenoise.c eggshell.c
    commandsManager.c processManager.c)
```

2.1.2 Explanation

For the eggshell, the CMake was used for compiling and the Linux bash for executing. Six c files were used, with eggshell.c containing the main, which is why it will be discussed last. The executable is therefore found inside /Eggshell/cmake-build-debug. linenoise.c will only be discussed where it was used as this is an external library and the use, not creation, of this library was required. Each of the other C files have headers associated with them, and therefore these will be discussed alongside their C files. As can be seen, each part of the program has it's own 'manager' in order to be able to split the problem into smaller pieces.

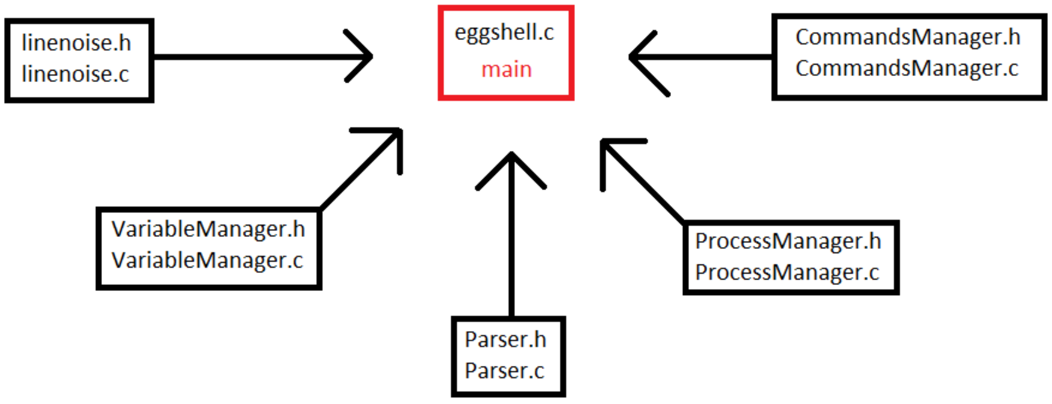


Figure 2.1: Hierarchy of eggshell, with the main and it's sub-parts

2.2 Variable Manager

2.2.1 variableManager.h

```c
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

#define MAX_ARGUMENTS 64
#define VARIABLE_LENGTH 512
#define NUMBER_OF_SHELL_VARIABLES 8

struct Variables{
    char key[VARIABLE_LENGTH], value[VARIABLE_LENGTH];
};

struct Variables** variables;
int variableAmount;

void setVariable(char* varName, char* newCharValue);
char* getVariable(char* varName);

void setUpShellVariables(char* shellNameWithDotSlash);

void removeSpaces(char* string);
char* trimFirstLetter(char* string);
```

2.2.2 Brief Explanation

The aim of this manager was to be able to abstract the main program from having to implement it's own variable interface. It contains a way of storing the variables, adding more variables and editing existing variables. A way to initialise variables which needed more complex methods was also implemented (setUpShellVariables) and any string manipulation is also done through this manager.

2.2.3 Detailed Explanation

First, some preprocessed variables are set in the header file. These are used throughout the program. A struct *Variables* contains the variables as key (name) and value (contents) pairs. *variables* contains a dynamic array of pointers to the struct *Variables*, in order to hold as many shell variables as needed. *variableAmount* is a counter to keep track of how many shell variables there are, so it is incremented each time a new variable is added.

setVariable takes in a name and value for a variable. If the name does not already exists, it adds a new one to the dynamic array, otherwise it modifies the existing variable's value.

getVariable is used to return the value of the variable, given its name/key. This performs a linear search trough the dynamic array. It returns *NULL* if the variable is not found.

setUpShellVariables creates or edits the variables *CWD* and *SHELL*. It uses *getcwd* to get the working directory of the program and where it was initially launched, then sets this as the value to the variable *CWD*. For the *SHELL* variable, the shell's name (first argument given to launch the shell from bash) is added to the *CWD*. The *TERMINAL* variable is set to the bash's tty. So for example, *CWD* could be set to */home/USERNAME/Desktop*, *SHELL* is then set to */home/USERNAME/Desktop/eggshell* and *TERMINAL* could be set to */dev/pts/2*.

trimFirstLetter is used for string manipulation to return a string with it's first letter removed. Buffering techniques are used for easier manipulation and correct output.

removeSpaces take a string parameter, and first removes the strings' initial spaces by ignoring the spaces and putting the rest of the characters in a buffer, then sets the character after the last alphanumeric character as the null character \0, so that spaces are only removed from the beginning and end of the string, while middle spaces are retained.

2.3 Parser

2.3.1 parser.h

```c
#include <string.h>
#include "variableManager.h"

void parseStringBy(char *string, char* parseString, char** returnArgs);
void parseStringByString(char *string, char* parseString, char**
    returnArgs);
```

2.3.2 Explanation

The parser is used to split strings into smaller strings. In order to split the string using an array of character delimiters, **parseStringBy** is used in conjunction with *strtok*. Meanwhile **parseStringByString** parses a string by another string. This was created for parsing <<< and >>. Both of these methods take a string to be parsed, a delimiter (or multiple delimiters in the case of the first function) and a pointer to arguments whose contents will be changed.

2.4 Process Manager

2.4.1 processManager.h

```
#include <stdlib.h>
#include <wait.h>

int* processes;
int processAmount;

void addProcess(int newPID);
int getLastProcess();
void removeLastProcess();
void removePID(int pid);
void waitForChild();
```

2.4.2 BriefExplanation

This manager takes care of how processes are stored. A dynamic array of integers *processes* is used to store the processes' PIDs as a stack. *processAmount* is increased each time a process is added, and decreased when a process is removed. The method to wait for the children and remove them from the stack are also done by this manager. A process can be removed from the array directly (example: if it was closed externally while in the background, rather than trough eggshell).

2.4.3 Detailed Explanation

addProcess is used when a new process is executed. The *processes* dynamic array is enlarged, the new PID is added to the top of the stack and *processAmount* is incremented by 1.

getLastProcess returns the process at the top of the stack.

removeLastProcess does the exact opposite of **addProcess**.

waitForChild is used when the parent executes or puts a child process in the foreground and needs to wait until it is finished, interrupted or stopped so that the parent can continue. Since the process may be stopped or interrupted by a signal, *WUNTRACED* is used. The variable *EXITCODE* is set to the status of the child after the parent waits for it.

removePID is like **removeLastProcess** but instead it removes a specific PID from the stack, which is treated as an array in this case.

2.5 Commands Manager

2.5.1 commandsManager.h

```c
#include <stdbool.h>
#include <sys/wait.h>

void printCommand(char** args);
void chdirCommand(char* newDir);
void variableModificationCommand(char* line);
void allCommand();
void sourceCommand(char* filename, void recursiveCall(char* line));
void forkChild(char** args, int* inputPipe, int* outputPipe, bool readFromPipe, bool writeToPipe);
```

2.5.2 Brief Explanation

This manager was used for executing both internal and external commands in eggshell. The first five displayed above are the internal commands, and are named appropriately. **variableModificationCommand** refers to the changing of the variables by the user during the user (ex. *$NEW-VARIABLE = CONTENT*). **forkChild** refers to when command is not recognized as an internal one and it instead tries to execute an external commands using the paths given.

2.5.3 Detailed Explanation

printCommand is called whenever *print* is the first argument. It outputs the string in front of it to the current *STDOUT*. As parameters, it takes the arguments it needs to print. The way it works is that if the first character of an argument is a '"', then it sets the flag *duringText* as true, and removes the '"' from the argument. If an argument ends with '"' , *duringText* is set to false and the final character of that argument is set to the NULL character *\0* instead of '"'. If an argument starts with '$' and *duringText* is false, then the variable with that name is searched and the argument is replaced with it's value (or *NULL* if it is not found. In any other case, the argument itself is printed with no alteration. If a '"' is found in the middle of a word, however, this is printed normally.

chdirCommand checks if the directory entered is correct and if it is, it changes the variable *CWD* accordingly.

variableModificationCommand is used to change or add a variable from the shell. If the line starts with a '$', the program assumes that a variable is being added or modified. It then parses the string by '=' and returns if no second argument is found. Otherwise, any spaces are removed, the '$' is removed from the first argument. Then a **setVariable** (discussed in subsection 2.2.3) is called with the first argument being the name and the second argument being the value. The second argument is also treated as a variable name if it starts with '$'.

forkChild takes 5 arguments. The arguments to be executed, 2 pipes (Input pipe and Output pipe) and 2 booleans to determine whether the executed child should read/write from the pipes given. The first thing the function does is go through the arguments, and if a '&' is found, it is replaced by *NULL* and *parentWait* is set to false, so that any other argument after '&' is ignored. If a '&' is not found, the parent will wait until the child exits, gets stopped or gets interrupted. The function then proceeds to fork a child, and while the parent waits (or continues on), the child searches trough *CWD* and *PATH* to search for a possible execution of the executable specified in the first argument using *execv* to execute the command with it's parameters. If execution fails, it goes to the next path, until a path manages to execute the command or there are no more paths to search and it outputs that it failed. The child also blocks *SIGINT* and *SIGTSTP*, so that these are not queued when the child is suspended. These signals are instead handled in the parent.

allCommand loops through the first *NUMBER_OF_SHELL_VARIABLES = 8* variables of the shell and outputs them.

sourceCommand reads line by line from a file and executes each one. The file is made sure to be in the current CWD. The command returns if no file is found, otherwise it goes through the list of commands in the file and sends them to *recursiveCall* one by one, as a line. This call will be the function to compute the line, in the main C file (eggshell).

2.6 Eggshell

2.6.1 Functions and Variables

```

void computeLine(char* line);
char* parseForOutputRedirection(char* line);
char* parseForInputReceival(char* line);
void parseForPiping(char* line);
void performCommand(char* line);
void initialiseShellVariables(char* zerothArgument);
void initialiseProcessesAndSignlalHandling();
void resetItemsAtEndOfCommand();
void resetIO();

void signalHandler(int signalNumber);

//Signal handler struct
struct sigaction sa;

//Used to store the default STDIN, OUT, ERR for reverting them after a
    command
int stdinFileDescriptor;
int stdoutFileDescriptor;
int stderrFileDescriptor;

//used for output and input redirection
FILE* outputFile;
FILE* inputFile;

//Only 2 pipes are used
int pipe1[2];
int pipe2[2];

//Bool to indicate wether child process should read/write
//Like pipes, 0 = read and 1 = write
bool IOint[2];

//An alternating bool to indicate which pipe is out/input, when we have
    multiple pipes
bool currentPipe = 0;

//A bool to indicate wether the parent should wait or not after forking a
    child
bool parentWait = true;
```

2.6.2 Brief Explanation

This is the main C file, used to do the main parsing and detection of the input line trough linenoise. It makes use of all the other smaller c files, which were the building blocks for this C file. For this class however a top down approach will be taken, instead of the bottom up.

Explaining the program step by step briefly, the program first parses the program the check for output redirection by checking for '>>' or '>' and sets *STDOUT* accordingly , then parses to check for input redirection and sets *STDIN* accordingly. After that, it parses for any pipes, and uses pipes if any are found. Each command between the pipes (if there are any) are computed individually. (For a detailed example, check out the **computeLine** subsection of the next section below)

Signals are handled trough *sigaction* and the signal handler send the appropriate signals to the child, while making use of the process stack in process manager.

2.6.3 Detailed Explanation

The **main** function sets some initial variables then loops indefinitely. During this loop, linenoise takes input line by line and processes each line accordingly. Null checking is performed to prevent crashes. Linenoise is also used for history functionality (being able to press the up and down arrow for previous and next typed commands).

initialiseProcessesAndSignlalHandling sets the processes amount to 0, allocates memory for the processes' PID stack and the signal handling for *SIGINT*, *SIGTSTP* and *SIGCHLD*.

computeLine parses the line given first by output redirection, then input redirection, then piping is checked for and a reset is then set in preparation for the next command. For an example, if the following command is entered: '*sort* | *figlet* < *input.txt* >' *output.txt* (no spaces needed due to the **removeSpaces** method discussed in subsection 2.2.3) the following steps would occur:

- *STDOUT* and *STDERR* are set to *output.txt* and 'line' is set to *sort* | *figlet* < *input.txt*
- Then *STDIN* is set to *input.txt* and line is now set to *sort* | *figlet*
- The line is now parsed by pipes and first *sort* is executed, using input from *input.txt*. It's output is then put to a pipe, from which *figlet* reads and then outputs to *output.txt*.

How each of the methods work is discussed below, this was a brief overview on how the program handles a single line of input with I/O redirection and pipes.

parseForOutputRedirection is responsible for handling '>' (write) and '>>' (append). It checks the given line and parses it through these. It first parses for append, since parsing with '>' before '>>' would return for write when an append is required, since they are the same character. If a second argument is returned from the parsing, this means that there was an output redirection, so a file is opened in append or write mode and *STDOUT* and *STDERR* are redirected to the specified file (if one is specified). *args[0]* is returned, meaning the part of the line before the output redirection symbol.

parseForInputReceival does the same thing as **parseForOutputRedirection** except this time it uses '<<<' for here string and '<' for file input. *STDIN* is redirected to the file. For here string, a temporary file is opened in 'wr' mode using *tmpfile*, the here string is written to the file, the file is re-winded and then passed as input to the command. Note: For both file input and file output, the files or temporary files are closed at the end of the command.

parseForPiping first gets a number of arguments and an argument count. The first command is set to output to the first pipe if and only if there is at least one '|'. Only two pipes need to be used, so that the first command outputs to the first pipe, then the second command gets it's input from the first pipe and outputs to the second, then the third opens up the first pipe again, gets its input from the second pipe and outputs to the first pipe... until the last command outputs to the current *STDOUT*. This switching between pipes 1 and 2 is done trough the boolean *currentPipe*. The reset of standard input and output as well as resetting *currentPipe* is only done at the end after all commands have been executed, so that next time, this starts from pipe 1 again. *IOint* is used to determine whether the process should get it's input or output to or from the pipe.

performCommand checks the first argument and sees if it matches with an internal command, otherwise it forks an external command with the first argument. The commands manager is used heavily here. It is not used for *fg* and *bg*. *fg* sends *SIGCONT* to the process at the top of the stack and waits for it, printing an error message if there are no processes. *bg* does the same thing as *fg*, except that it does not wait for the process. *jobs* is a method used for displaying the amount of processes on the stack at the time of calling.

resetItemsAtEndOfCommand resets standard input and output, closes any open files, resets pipes input and output and sets the current pipe to it's default value, so that it restarts when pipes are used again later on.

signalHandler is the handler for *SIGCHLD*, *SIGINT* and *SIGTSTP*. *SIGCHLD* may be called when a child process exits externally (Example when a web bowser is closed through the *X* button), rather than trough *CTRL+C* in the shell, so when this is received, it waits for every child process and kills and zombies as well as removing them from the process table. *SIGINT* and *SIGTSTP* are not normally received during the linenoise of the eggshell, since linenoise handles them itself, so the only way these can be received are during a child process. So these two send *SIGKILL* and *SIGSTOP* respectively to the child processes.

Note: The reason for child processes ignoring *SIGINT* and *SIGTSTP* is because when a signal is sent to a child process, it is sent to the parent and all it's children. For example, if process A is executed and stopped, then process B is executed and given the interrupt signal, this signal is sent to the parent of both A and B, to B and to A. This results in *SIGINT* being queued in the signal queue for process A, so that when this continues it is immediately terminated. Hence the use of the handlers to send *SIGKILL* and *SIGSTOP*.

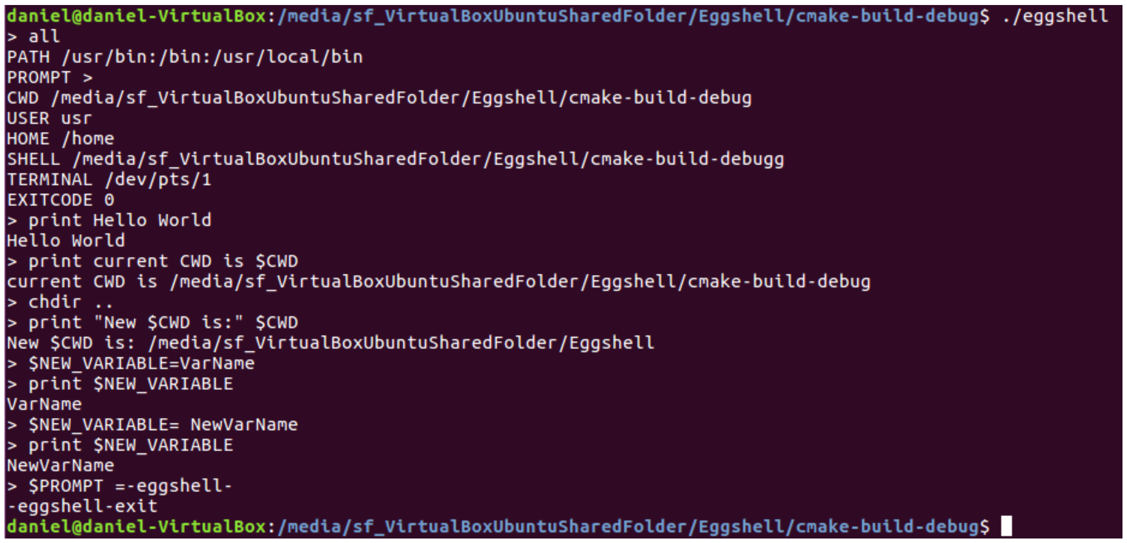
Chapter 3

Tests and Results

3.1 Internal Commands

For this section, the internal commands *all*, variable modification, *print*, *chdir*, *source* and *exit* will be shown.

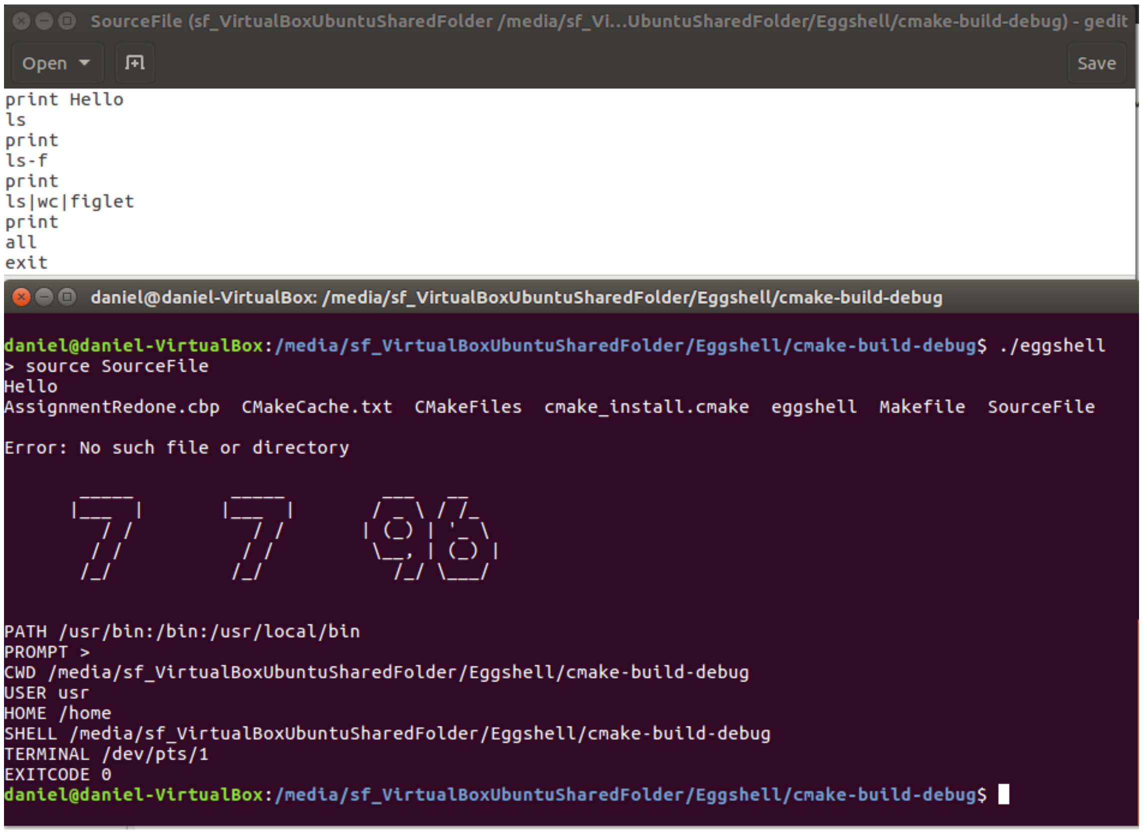
The following figure shows all the variations of printing, creating and editing a new variable (Note the spaces between variable name, value and the '='), editing an internal command (*PROMPT*), changing the *CWD* using *chdir*, the all command displaying the initialised shell variables and the exit command. Another thing that works that is not shown below is when doing *$VARIABLE1 = $VARIABLE2*, it will copy the value of *VARIABLE2* to *VARIABLE1*.



```
daniel@daniel-VirtualBox:/media/sf_VirtualBoxUbuntuSharedFolder/Eggshell/cmake-build-debug$ ./eggshell
> all
PATH /usr/bin:/bin:/usr/local/bin
PROMPT >
CWD /media/sf_VirtualBoxUbuntuSharedFolder/Eggshell/cmake-build-debug
USER usr
HOME /home
SHELL /media/sf_VirtualBoxUbuntuSharedFolder/Eggshell/cmake-build-debugg
TERMINAL /dev/pts/1
EXITCODE 0
> print Hello World
Hello World
> print current CWD is $CWD
current CWD is /media/sf_VirtualBoxUbuntuSharedFolder/Eggshell/cmake-build-debug
> chdir ..
> print "New $CWD is:" $CWD
New $CWD is: /media/sf_VirtualBoxUbuntuSharedFolder/Eggshell
> $NEW_VARIABLE=VarName
> print $NEW_VARIABLE
VarName
> $NEW_VARIABLE= NewVarName
> print $NEW_VARIABLE
NewVarName
> $PROMPT =-eggshell-
-eggshell-exit
daniel@daniel-VirtualBox:/media/sf_VirtualBoxUbuntuSharedFolder/Eggshell/cmake-build-debug$
```

Figure 3.1: Screen shot of the internal commands all, variable modification, print, chdir and exit

The below figure shows the *source* command. The file used is called **SourceFile** and will be included with the code in the directory */Eggshell/cmake-build-debug* for any further possible use.



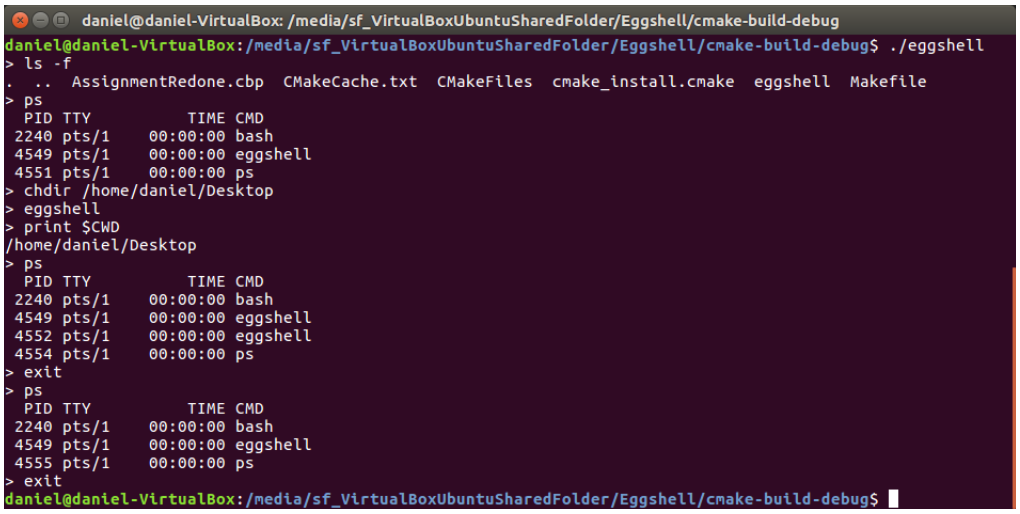
```
print Hello
ls
print
ls-f
print
ls|wc|figlet
print
all
exit
```

```
daniel@daniel-VirtualBox:/media/sf_VirtualBoxUbuntuSharedFolder/Eggshell/cmake-build-debug$ ./eggshell
> source SourceFile
Hello
AssignmentRedone.cbp  CMakeCache.txt  CMakeFiles  cmake_install.cmake  eggshell  Makefile  SourceFile

Error: No such file or directory

       _____          _____          ___    __
      |___  |        |___  |        / _ \ / /_
         / /            / /        | (_) | '_ \
        / /            / /          \__, | (_) |
       /_/            /_/             /_/ \___/


PATH /usr/bin:/bin:/usr/local/bin
PROMPT >
CWD /media/sf_VirtualBoxUbuntuSharedFolder/Eggshell/cmake-build-debug
USER usr
HOME /home
SHELL /media/sf_VirtualBoxUbuntuSharedFolder/Eggshell/cmake-build-debug
TERMINAL /dev/pts/1
EXITCODE 0
daniel@daniel-VirtualBox:/media/sf_VirtualBoxUbuntuSharedFolder/Eggshell/cmake-build-debug$
```

Figure 3.2: Screen shot of the internal command source. This uses external commands which will be shown in further detail later in this section

3.2 External Commands

In this section, external commands are shown. Note that for this, besides searching in the *PATH*, the program also checks in the *CWD*. Another eggshell executable was put on the desktop, and executed as an external command. Notice how *ps*, shows two eggshells running, until exited.

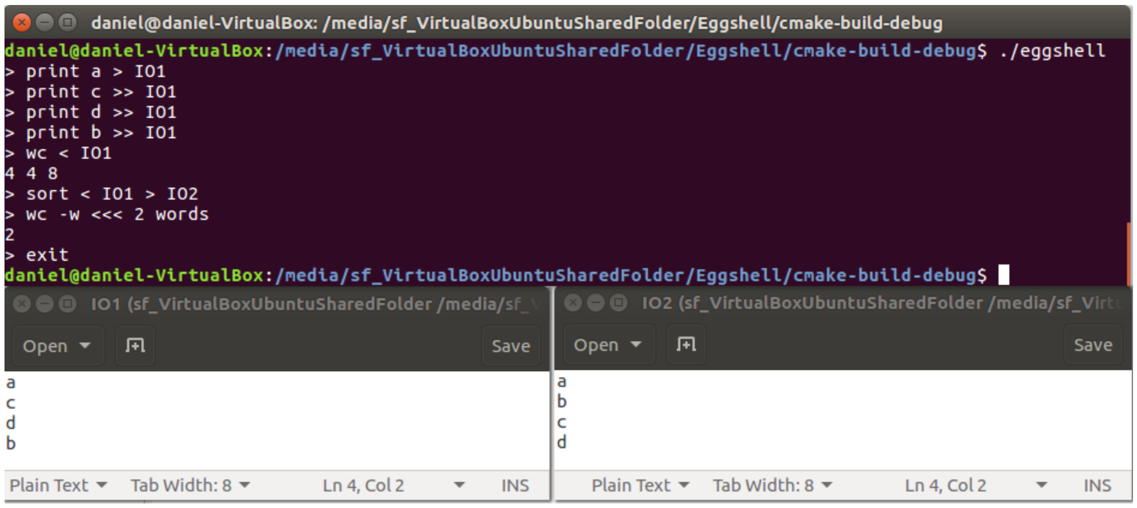


```
daniel@daniel-VirtualBox: /media/sf_VirtualBoxUbuntuSharedFolder/Eggshell/cmake-build-debug
daniel@daniel-VirtualBox:/media/sf_VirtualBoxUbuntuSharedFolder/Eggshell/cmake-build-debug$ ./eggshell
> ls -f
.  ..  AssignmentRedone.cbp  CMakeCache.txt  CMakeFiles  cmake_install.cmake  eggshell  Makefile
> ps
  PID TTY          TIME CMD
 2240 pts/1    00:00:00 bash
 4549 pts/1    00:00:00 eggshell
 4551 pts/1    00:00:00 ps
> chdir /home/daniel/Desktop
> eggshell
> print $CWD
/home/daniel/Desktop
> ps
  PID TTY          TIME CMD
 2240 pts/1    00:00:00 bash
 4549 pts/1    00:00:00 eggshell
 4552 pts/1    00:00:00 eggshell
 4554 pts/1    00:00:00 ps
> exit
> ps
  PID TTY          TIME CMD
 2240 pts/1    00:00:00 bash
 4549 pts/1    00:00:00 eggshell
 4555 pts/1    00:00:00 ps
> exit
daniel@daniel-VirtualBox:/media/sf_VirtualBoxUbuntuSharedFolder/Eggshell/cmake-build-debug$
```

Figure 3.3: Screen shot showing external commands being used

3.3 Input/Output Redirection

Regarding IO redirection, the output is first parsed for output and then for input, so when it comes to how the commands should be if both output and input redirection need to be used in the same command, the user should write the command as shown below, ie *sort <IO1 > IO2*. Note in the example how IO2 is the sorted version of IO1.



```
daniel@daniel-VirtualBox: /media/sf_VirtualBoxUbuntuSharedFolder/Eggshell/cmake-build-debug
daniel@daniel-VirtualBox:/media/sf_VirtualBoxUbuntuSharedFolder/Eggshell/cmake-build-debug$ ./eggshell
> print a > IO1
> print c >> IO1
> print d >> IO1
> print b >> IO1
> wc < IO1
4 4 8
> sort < IO1 > IO2
> wc -w <<< 2 words
2
> exit
daniel@daniel-VirtualBox:/media/sf_VirtualBoxUbuntuSharedFolder/Eggshell/cmake-build-debug$
```

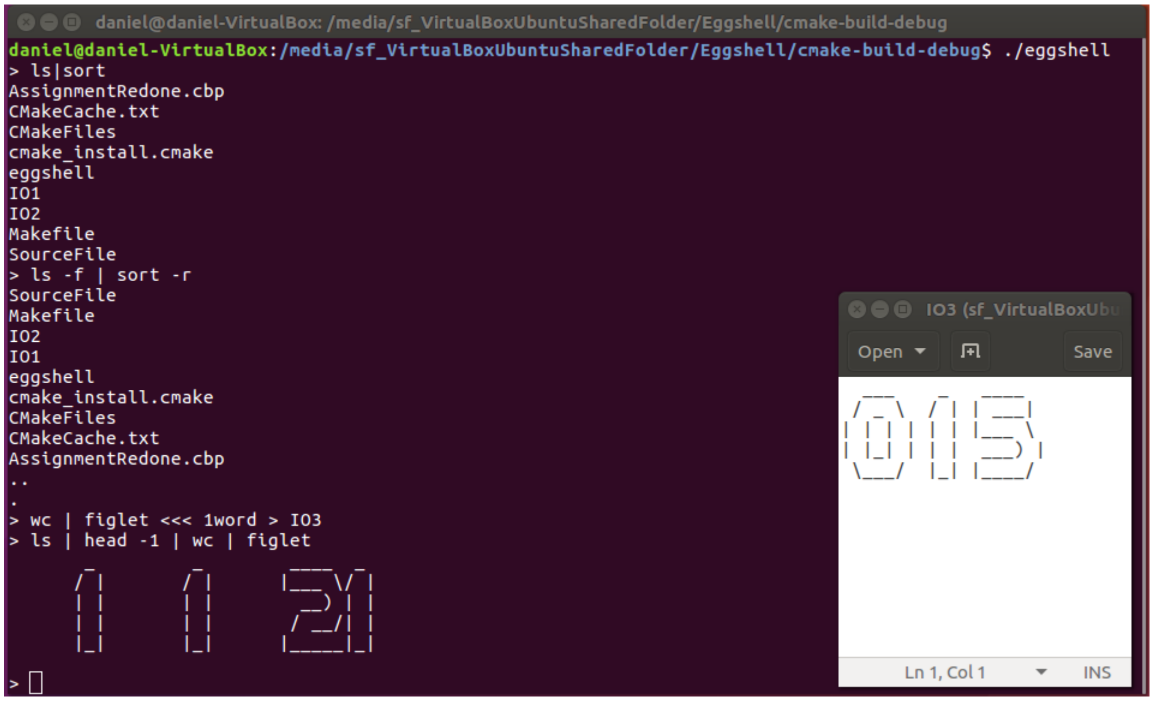
```
IO1 (sf_VirtualBoxUbuntuSharedFolder /media/sf_V
a
c
d
b
```

```
IO2 (sf_VirtualBoxUbuntuSharedFolder /media/sf_Virt
a
b
c
d
```

Figure 3.4: Screen shot of IO redirection with the files at the end of the commands

3.4 Piping

In order to pipe, the program performs each command in order. Pipes may also take input and output redirection, as can be seen below. Piping may be done between an infinite number of processes (or at least until the maximum number of characters in a line is reached).



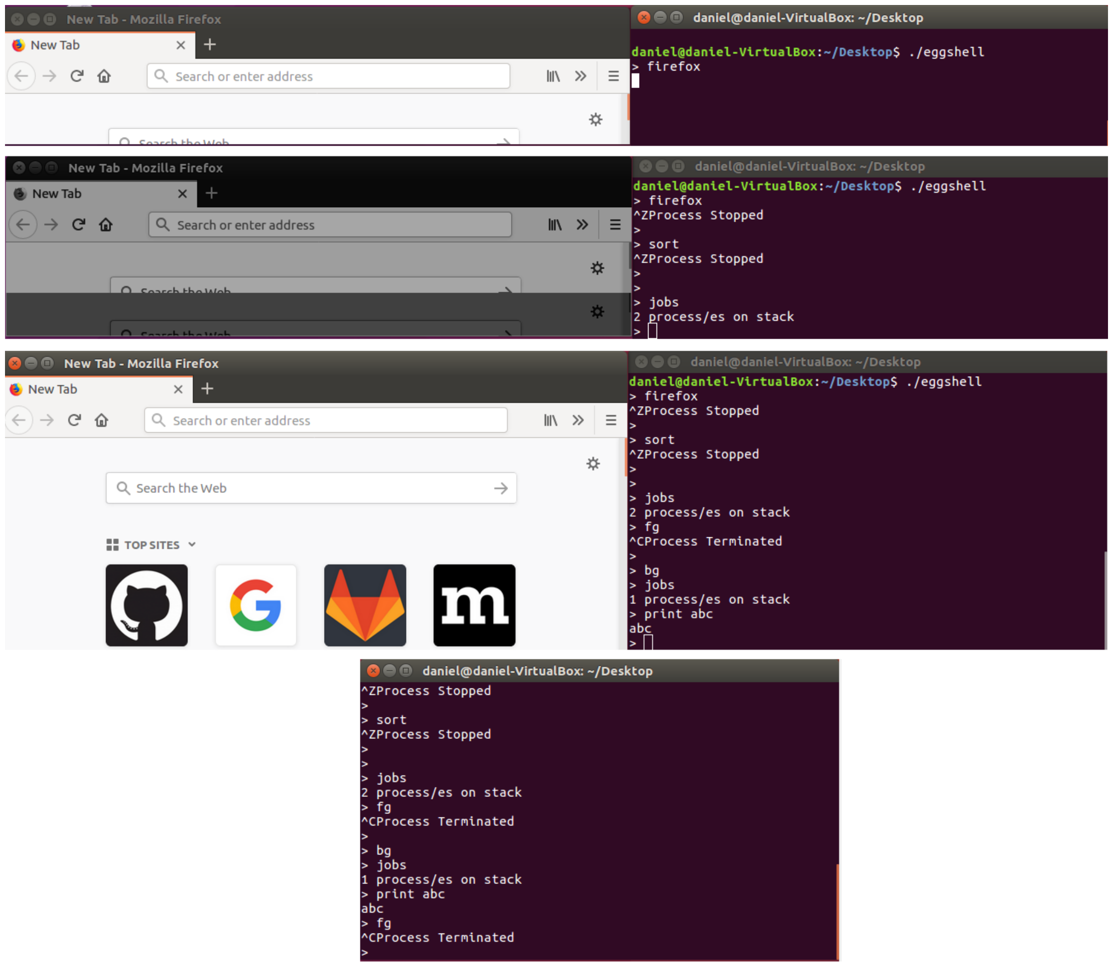
```
daniel@daniel-VirtualBox: /media/sf_VirtualBoxUbuntuSharedFolder/Eggshell/cmake-build-debug
daniel@daniel-VirtualBox:/media/sf_VirtualBoxUbuntuSharedFolder/Eggshell/cmake-build-debug$ ./eggshell
> ls|sort
AssignmentRedone.cbp
CMakeCache.txt
CMakeFiles
cmake_install.cmake
eggshell
IO1
IO2
Makefile
SourceFile
> ls -f | sort -r
SourceFile
Makefile
IO2
IO1
eggshell
cmake_install.cmake
CMakeFiles
CMakeCache.txt
AssignmentRedone.cbp
..
.
> wc | figlet <<< 1word > IO3
> ls | head -1 | wc | figlet
      _       _      ____  _ 
     / |     / |    |___ \/ |
     | |     | |      __) | |
     | |     | |     / __/| |
     |_|     |_|    |_____|_|

> 
```

```
IO3 (sf_VirtualBoxUbu
  ___  _ ____  
 / _ \/ | ___| 
| | | | |___ \ 
| |_| | |___) |
 \___/|_|____/ 
```

Figure 3.5: Screen shot of Piping used in the eggshell

3.5 Signals

For signals, multiple processes can be stacked. Signals are blocked by the child process, so the parent sends the appropriate signals to the children to suspend or interrupt them. However *fg* and *bg* can only be sent to the top process on the stack. For the example, Firefox is used as it is a great visual representation. . In the fourth and last image, Firefox is not shown because it is closed.


New Tab - Mozilla Firefox
New Tab
Search or enter address
Search the Web
TOP SITES
daniel@daniel-VirtualBox: ~/Desktop

```
daniel@daniel-VirtualBox:~/Desktop$ ./eggshell
> firefox
```

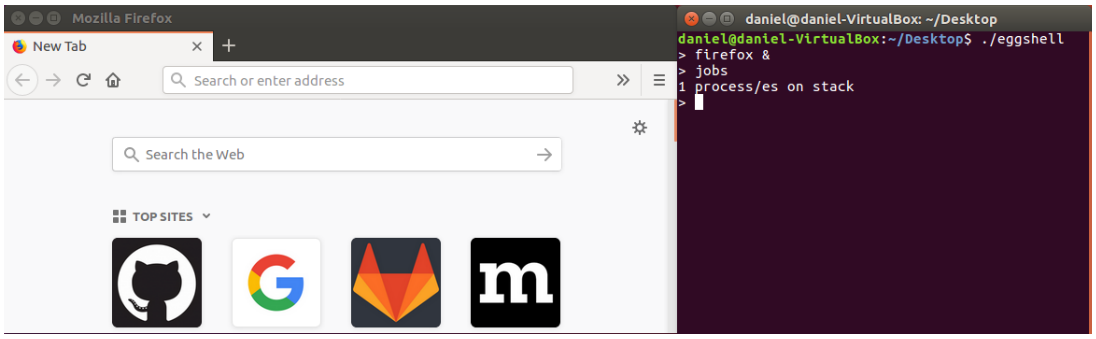
```
daniel@daniel-VirtualBox:~/Desktop$ ./eggshell
> firefox
^ZProcess Stopped
>
> sort
^ZProcess Stopped
>
>
> jobs
2 process/es on stack
>
```

```
daniel@daniel-VirtualBox:~/Desktop$ ./eggshell
> firefox
^ZProcess Stopped
>
> sort
^ZProcess Stopped
>
>
> jobs
2 process/es on stack
> fg
^CProcess Terminated
>
> bg
> jobs
1 process/es on stack
> print abc
abc
>
```

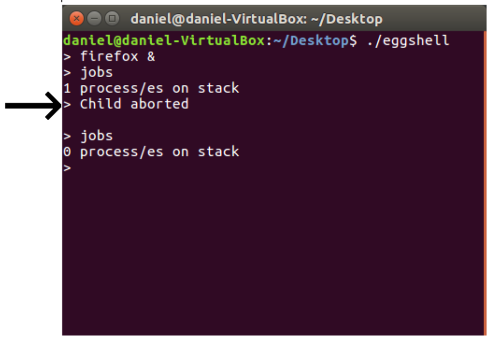
```
^ZProcess Stopped
>
> sort
^ZProcess Stopped
>
>
> jobs
2 process/es on stack
> fg
^CProcess Terminated
>
> bg
> jobs
1 process/es on stack
> print abc
abc
> fg
^CProcess Terminated
>
```

Figure 3.6: Four screen shots showing different states of Firefox at different stages of the program after different commands and shows how processes are stored

The following shows a case where a process in the background is terminated externally (when pressing the *X* button on firefox). In the case below, when Firefox exits, the message *child aborted* is displayed.



```
daniel@daniel-VirtualBox:~/Desktop$ ./eggshell
> firefox &
> jobs
1 process/es on stack
> 
```



```
daniel@daniel-VirtualBox:~/Desktop$ ./eggshell
> firefox &
> jobs
1 process/es on stack
> Child aborted

> jobs
0 process/es on stack
>
```

Figure 3.7: Two screen shots showing when the Firefox is launched in the background then terminated externally

Chapter 4

Known Issues and Improvements

Although every part of the eggshell was implemented with the best effort, given the time limit, certain things could have been designed better or with more functionality and certain nooks could have been better fixed. These are the following:

- The internal command *print* takes no file input. An extra feature would be for this to do the functionality that the external command *cat* does.
- Piping only works for external commands.
- When print is given for example: *print $CWD!*, due to the exclamation point at the end (could be a comma or fullstop instead of the exclamation point, same thing still applies), this variable is not changed to it's value, but stays the same.
- Certain processes such as Firefox, do not close when the eggshell is closed. While an exit handler and *atexit* to close all processes on the stack would have been a very quick fix, it was noticed that the Linux bash itself does not close Firefox when closed, so this was omitted from the implementation. Note: processes like *sort* are still closed correctly.
- During variable creation, if the value (string after '=') has any spaces, these will be removed and the variable will be joined. Example: being given *$NEWVAR = VAL UE*, will assign 'VALUE' to *NEWVAR*.
- if print is given for example *print "hello   world"*, *hello world* will be printed. Interestingly, *echo* from the bash also removes any double spaces.
- A feature that would have been nice to add was that jobs store the name of the process as welll, rather than just the PID, so that when *jobs* is typed in, the entire list of processes is displayed. Then the user could choose to start any process they wants in the background or foreground, rather than just the last one on the stack. However, in order to see the processes, the external command *ps* could always be used.
- in order to initialise or change the value of variables, one cannot simply write $a = c$, but the *$* must be used before the a, ie *$a = c*.
- Sometimes, when it exits trough *CTRL+C*, Firefox returns an error, and the next time it is opened it requests to be opened up in safe mode (shown in the video).

Chapter 5

Conclusion

In conclusion, this assignment was very helpful for familiarising with the C programming language and how the Linux OS works in terms of how processes are stored and executed, how inter-process communication works and how the world of signalling is in the UNIX environment. It was also useful in learning how to use the Linux bash and how it could be a more powerful utility than using the more user friendly GUI.

While the eggshell provided in this document is not perfect, it emulates the eggshell in a lot of ways and improves in some ways. An improvement would be for example, in the normal bash, doing $a = c$ would result in an error, since it cannot handle spaces. Meanwhile, the eggshell provided handles spaces when doing *\$a = c*. While extra features and complete emulation/improvements to the Linux bash would have been lovely to implement, time constraints were present so that only the assigned requirements alongside some little extra functionality could be provided.