Introduction

A Unix shell is a command line interpreter that provides users with the ability to control the operation of the computer through text commands. The goal of this assignment is to implement `smash` (*sma*ll *sh*ell), a small and lightweight Linux shell, described in the second part of this document. This assignment will be evaluated on a number of criteria, including correctness, structure, style and documentation. Therefore, your code should do what it purports to do, be organised into functions and modules, be well-indented, well-commented and descriptive (no cryptic variable and function names), and include a design document describing your solution.

Overview

There are two parts to this document; in the first part, Unix shell fundamentals are briefly explored, using examples from the Bourne-again shell (Bash) to illustrate how certain `smash` features are meant to be implemented. The second part describes `smash` in terms of its cardinal components: command line interpreter, shell variables, input and output redirection and pipelines. Each component is broken down into its constituent features, which are sorted in order of suggested implementation, where possible.

Deliverables

Upload your source code (C files together with any accompanying headers), including any unit tests and additional utilities, through VLE by the specified deadline. Include makefiles with your submission which can compile your system. Make sure that your code is properly commented and easily readable. Be careful with naming conventions and visual formatting. Every system call output should be validated for errors and appropriate error messages should be reported. Include a report describing:

- the design of your system, including any structure, behaviour and interaction diagrams which might help;
- any implemented mechanisms which you think deserve special mention;
- your approach to testing the system, listing test cases and results;
- a list of bugs and issues of your implementation.

Do **not** include any code listing in your report; only snippets are allowed.

Record a video presentation of not more than ten minutes where you showcase and discuss your `smash` implementation. Alongside the demonstration, you are free to show snippets of code, discuss shortcomings and features, bugs and accomplishments. Upload the video presentation to your Google Drive and link to it from your report.

Part I

Shell Fundamentals

In an interactive shell, a command prompt informs the user that the shell is ready to accept commands; the user then proceeds to type in commands, getting back output in return:

```
host@user:~$ command arg0 arg1 arg2 ...
output from command
some more output
and even more output...
# Command prompt : ready to accept user input
host@user:~$
```

Commonly, user input takes the form of a `command` followed by a sequence of arguments (`arg0 arg1` and so on). These commands are categorised into *internal / builtin* commands – commands the shell knows about and can execute without any external dependencies (e.g. `cd`) – or *external* commands, which the shell knows nothing about. In fact, external commands are merely binary program images that live somewhere on the file system and the shell executes on behalf of the user. An external command may be specified through an absolute path (e.g. `/bin/ls`) or relative to some entry in the file system (e.g. `ls`). In the latter case, where ambiguity may arise, the shell employs a resolution strategy which iterates through a list of search paths in some predefined order, looking for the program binary every step along the way. These search paths are stored in one of a number of configuration variables the shell employs, called the `PATH`. These variables are alternatively known as environment variables. A typical path on a Unix-like system reads as follows:

```
$ echo $PATH
/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/opt/X11/bin
```

These variables may be modified by the user through an assignment statement:

```
$ PATH=/usr/bin:/bin:/home/ashina
$ echo $PATH
/usr/bin:/bin:/home/ashina
```

The statement overwrites the value of a variable with the right hand side of the assignment; if the variable doesn't exist, it is created.

```
$ echo $SOMEVAR

$ SOMEVAR=something
$ echo $SOMEVAR
something
```

The shell maintains a second tier of variables that are not exposed in the environment. These are referred to as shell variables. Environment and shell variables can be thought of as being defined at two different scoping levels; while environment variables are *global*, shell variables are *local* or *private* to the shell. Shell variables can be promoted or *exported* to environment scope:

```
$ SOMEVAR=something
# printenv should not output anything since SOMEVAR is not part of the environment
$ printenv SOMEVAR
# export adds the variable to the environment
$ export SOMEVAR
$ printenv SOMEVAR
something
```

When a program is launched, environment (but not shell) variables are made available to it. From within a C program, there are two ways by which these variables may be accessed: the `environ` external declaration

```
extern char **environ;
```

and the third argument of the `main` function:

```
int main(int argc, char **argc, char **env)
```

Additionally, programs launched by the shell have a number of pre-connected communication channels made available to them at start-up; these communication channels, or streams, exist to abstract I/O operations, irrespective of the actual device these channels are wired to. There are three such channels: *standard input (stdin)*, *standard output (stdout)*, and *standard error (stderr)*. Initially wired to the terminal, the shell provides redirection operators to override the default bindings of these streams: `<, <<<, >, >>`.

```
$ sort < myfile.txt # Redirect input from file
$ sort <<< 'one
  two
  three' # Use 'here strings' for input redirection
$ cpp myfile.c > myfile.i # Redirect output to file
$ webserver >> ws.log # Redirect and append output to file
```

Furthermore, distinct streams from multiple programs can be wired together into a pipeline, such that the output of one process becomes input to the next, forming a chain of producers and consumers. This is accomplished through the pipe `|` operator:

```
$ echo $PATH | tr : '\n' | grep usr
/usr/bin
/usr/sbin
/usr/local/bin
```

A shell returns control to the user once a program has done executing; programs that block the shell while executing are said to be running in the foreground:

```
$ webserver
Web server running
...
Web server terminating
# Shell prompt; user given control after completion
$
```

The shell can be made to return control to the user immediately after launching a program, even though the latter will not have terminated. Such programs are said to be running in the background; a shell launches a program as a background process by appending the background operator & at the end of the command string:

```
$ webserver &
[1] 34003
Web server running
# User given control immediately after launch
$
...
Web server terminating
[1]+ Done
$
```

The shell provides mechanisms for suspending or interrupting a process running in the foreground. Typically such a process is suspended on receiving `SIGTSTP`, triggered via Ctrl + Z, or interrupted when Ctrl + C are pressed and `SIGINT` is triggered. A suspended process may be resumed either as a foreground process, using `fg`, or as a background process, using `bg`.

```
$ webserver
Web server running
# Ctrl + C
Web server terminating
$
```

For further information on Unix shells, the reader is referred to the Linux `man` pages.

Part II

`smash`

In your assignment, you are to implement `smash`, a simple command line interpreter for the Linux operating system. There are six sections in the specification document: *Input, Tokenisation and Parsing*; *Variables*; *Internal Commands*; *External Commands*; *Input and Output Redirection*; and *Pipelines*.

Input, Tokenisation and Parsing

Throughout this assignment, we will use the following definitions:

`whitespace` a `space` or `tab` character

`token` a sequence of characters considered as a single unit by the shell

`metacharacter` a character that when unquoted, separates tokens; one of the following: `| ; < > space tab`

`control operator` a token that performs a control function; one of the following symbols: `; | newline`

`command` a sequence of optional variable assignments (*see Variables*) followed by `whitespace`-separated tokens and redirections (*see Input and Output Redirection*), and terminated by a `control operator`

`pipeline` a sequence of one or more commands separated by the control operator `|`; *see Pipelines*

(a) Implement command line input; the `linenoise` utility may be used as a replacement for `readline`. Example usage of the library is provided below:

```c
char *line;

while ((line = linenoise("prompt> ")) != NULL)
{
        // Echo output back
        printf("Input echo: [%s]\n", line);

        // Tokenise and parse command line
        ...

        // Free allocated memory
        linenoiseFree(line);
}
```

(b) Implement tokenisation of the command line: group together characters in the input string using `metacharacters` as separators.

(c) Quoting is used to remove the special meaning of certain characters to the shell. There are two quoting mechanisms in `smash`: the *escape character* and *double quotes*. Implement quoting such that:

i) A non-quoted backslash `\` is the *escape character*. It preserves the literal value of the following character.

ii) Enclosing characters in double quotes `"..."` preserves the literal value of each character within the quotes, with the exception of `\` and `$`.

iii) Each of the `metacharacters` listed above holds special meaning to the shell and must be quoted to represent itself.

(d) Expansion is performed on the command line after it has been split into `tokens`. In `smash` only one kind of expansion is performed: *variable* expansion (*see Variables*). Implement expansion such that:

i) The `$` character introduces variable expansion.

ii) The variable to be expanded may be enclosed in braces `{...}`, which are optional but serve to protect the variable to be expanded from characters immediately following it, to stop them from being interpreted as part of the name.

iii) The basic form of variable expansion is `$VARIABLE_NAME` or `${VARIABLE_NAME}`.

(e) Implement *quote removal* such that after expansions, all unquoted occurrences of the characters `\` and `"` that did not result from one of the above expansions are removed.

Variables

A shell variable is a character string to which some value is assigned. The name of a shell variable can contain letters (a-z, A-Z), numbers (0-9) or the underscore character (_), but cannot start with a number. Conventionally, Unix shell variables are expressed in UPPERCASE.

(a) The shell is expected to manage the following variables:

(i) `PATH` - The search path used to launch external commands; subpaths are delimited by a colon, e.g. `/usr/bin:/bin:/usr/local/bin`.

(ii) `PROMPT` - The string presented to the user to show that the shell is ready to accept command input, e.g. `smash-1.0 >`.

(iii) `CWD` - The current working directory; all file operations are relative to this path, e.g. `/home/student/cps1012`.

(iv) `USER` - The username of the current user, e.g. `student`.

(v) `HOME` - The home directory of the current user, e.g. `/home/student`.

(vi) `SHELL` - The shell of the current user; should be the absolute path of the `smash` binary, e.g. `/home/student/cps1012/bin/smash`.

(vii) `TERMINAL` - The name of the terminal attached to the current `smash` session; for example: `/dev/pts/1`.

(viii) `EXITCODE` - The exit code returned by the last program run by the shell.

Note that shell variables should be populated as the shell starts up.

(b) A user should be allowed to modify the content of shell variables or create new ones by using an assignment statement, `VARIABLE=VALUE`. Note that neither `VARIABLE` nor `VALUE` are separated from `=` by `whitespace`:

```
$ USER=billyxatba
$ HOME=/home/billy
$ PATH=/usr/bin:/bin
$ A=1 B=2 C=3 D=4
```

(c) The shell should recognise the `$` special character and perform variable expansion accordingly, where the variable name is textually replaced by its value:

```
$ DIRNAME="/home/lowbattery/controller/"
$ echo DIRNAME
DIRNAME
$ echo $DIRNAME
/home/lowbattery/controller/
$ echo $DIRNAME1

$ echo ${DIRNAME}1
/home/lowbattery/controller/1
```

Some more examples using quoting:

```
$ NAME=One of these days
bash: of: command not found
$ NAME=One\ of\ these\ days
$ echo $NAME
One of these days
$ NAME="you can have them."
$ echo $NAME
you can have them.
$ NAME=Any\ colour\ you\ like", they are all blue."
$ echo $NAME
Any colour you like, they are all blue.
$ NAME="Luffy" NICK=Straw\ Hat NAME2="Monkey D. $NAME"
$ DESC="$NAME2 also known as \"$NICK $NAME\" and commonly as \"$NICK\""
$ DESC="$DESC, is the main protagonist of the manga and anime, One Piece."
$ echo $DESC
Monkey D. Luffy also known as "Straw Hat Luffy" and commonly as "Straw Hat",
 is the main protagonist of the manga and anime, One Piece.
```


PTO

Internal Commands

This section describes the `smash` internal commands. Your implementation should trap and handle errors; when a command fails, the user should be made aware through an appropriate error message.

(a) Implement the `exit` command to quit the shell. By default exit returns with 0 unless a value is specified.

Hint: You can use the `exit()` function or `return` from the `main` function of `smash` with an exit code.

(b) Implement the `echo` internal command, to print text to standard output.

(i) Echo text to standard output:

```
$ echo hello, this is a test!
hello, this is a test!
```

(ii) Print the value of a shell variable:

```
$ echo $PATH
/usr/bin:/bin:/home/student
$ echo $USER
student
```

(iii) Print combined text and shell variable values (except in the case of quoted characters):

```
$ echo hello $USER, welcome to smash!
hello student, welcome to smash!
$ echo "hello $USER, welcome to smash!"
hello student, welcome to smash!
$ echo hello \$USER, welcome to smash!
hello $USER, welcome to smash!
$ echo \"hello $USER, welcome to smash!\"
"hello student, welcome to smash!"
```

Hint: You can use the `fprintf()` function to output to a specified output stream.

(c) Implement the `cd` command to change the current working directory (*see Variables*, `CWD`).

```
$ echo $CWD
/home/student/cps1012
$ cd ..
$ echo $CWD
/home/student
$ cd /home/student/cps1012/bin
$ echo $CWD
/home/student/cps1012/bin
```

Hint: You can use the `chdir()` function to change the current directory programmatically.

(d) Implement the `showvar` command to print shell variables, in key-value pairs, to standard output.

(i) Print all shell variables to standard output, one pair per line:

```
$ showvar
PATH=/usr/bin:/bin:/home/student
PROMPT=smash-1.0>
SHELL=/home/student/cps1012/bin/smash
USER=student
HOME=/home/student
CWD=/home/student/cps1012
TERMINAL=/dev/pts/1
```

(ii) Print a specific shell variable:

```
$ showvar PROMPT
PROMPT=smash-1.0>
```

(d) Implement the `export` command to promote a shell variable to environment scope. Promoted variables are also inherited by the environment of programs launched through the `smash`.

```
$ TEST="This is a test."
$ echo $TEST
This is a test.
$ printenv TEST
$ export TEST
$ printenv TEST
This is a test.
```

Hint: You can use the `setenv()` function to create new environment variables; alternatively, if a variable exists, setting the `overwrite` argument will update the value of the variable in the environment.

(e) Implement the `unset` command to delete an existing variable, from the shell and environment both.

```
$ TEST="This is a test."
$ export TEST
This is a test.
$ unset TEST
$ echo $TEST
```

Hint: Use the `unsetenv()` function to delete a variable name from the environment. If the variable does not exist in the environment, then the function succeeds, and the environment is unchanged.

(f) Implement the `showenv` command to print environment variables, in key-value pairs. This command is similar to `showvar`, but instead of outputting shell variables, it outputs environment variables and their values to the standard output stream. Like `showvar`, one can print a specific variable by passing its name as an argument.

```
$ showenv TERM
TERM=xterm-256color
```

Hint: Use the `getenv()` function to search the environment list for the given variable name and return a pointer to the corresponding value string.

(g) Implement a directory stack that the shell can use to remember directories. Directories are added to the list with the `pushd` command, while the `popd` command removes directories from the list in Last-In First-Out order. The current directory is always the first directory in this stack. The `dirs` command should make `smash` print the directory stack to standard output. In summary:

(i) Implement a directory stack to store directories; the current working directory is always the first directory in this stack.

(ii) Implement the `pushd` command to add directories to the stack. Make sure that the directory exists before pushing it on the stack.

```
$ pwd
/home/pizzacutter
$ pushd /home/moonlight
/home/moonlight /home/pizzacutter
$ pwd
/home/moonlight
$ pushd /home/zweihander
/home/zweihander /home/moonlight /home/pizzacutter
$ pwd
/home/zweihander
```

(iii) Implement `popd` to remove a directory from the stack. The stack should never be empty; it should contain at least one entry.

```
$ pwd
/home/pizzacutter
$ pushd /home/moonlight
/home/moonlight /home/pizzacutter
$ pwd
/home/moonlight
$ popd
/home/pizzacutter
$ pwd
/home/pizzacutter
```

(iv) Implement the `dirs` command to print the directory stack.

```
$ pwd
/home/pizzacutter
$ pushd /home/moonlight
/home/moonlight /home/pizzacutter
$ pushd /home/zweihander
/home/zweihander /home/moonlight /home/pizzacutter
$ dirs
/home/zweihander /home/moonlight /home/pizzacutter
```

(v) The current working directory and the first entry in the directory stack should always mirror each other. Make sure that this constraint is preserved after every `pushd`, `popd` and `cd` operation. When the shell launches, the directory stack is initially populated with the user's home directory.

Hint: Keep the directory stack in mind when implementing the `cd` command. Changes to the current working directory are reflected in the first entry of the directory stack; moreover, adding or removing directories from the stack affects the current working directory.

(i) Implement the `source` command to equip `smash` with basic scripting functionality. The command takes a single argument, specifying the name of the script file. It then proceeds to open this file and execute commands from it, one line at a time, until the end-of-file is encountered. Each executed command should behave exactly in the same way as if it were typed in.

Hint: You should abstract command input from parsing and execution, to make the implementation of the `source` command more straightforward - a change of input modality.

External Commands

The shell assumes commands not included in the list above to be external commands, and will search for a matching program binary to launch it as a separate process.

(a) Use the *fork-plus-exec* pattern to execute unrecognised internal commands as external commands.

(b) Use the system search path (`PATH`) to look for command binaries.

(c) Verify that programs launched from `smash` contain the `smash`-specific shell variable definitions (e.g. `TERMINAL` and `CWD`).

Input and Output Redirection

A very powerful feature of Unix shells is the ability to redirect output and input to and from files. Likewise, `smash` supports a number of redirection operators:

(a) Implement output redirection operators `>` and `>>`:

`command` `>` `file` - Redirect the output from `command` into `file`.

`command` `>>` `file` - Redirect the output from `command` into `file`, appending to its current contents.

(b) Implement the input redirection operator `<`:

`command` `<` `file` - Use the contents of `file` as input to `command`.

Note that `command` refers to an arbitrary internal or external command, with zero or more arguments. Likewise, `file` refers to an arbitrary file.

Pipelines

In Unix-like computer operating systems, a command pipeline is a sequence of processes chained together by their input and output streams, so that the output of one process (`stdout`) feeds directly into the input (`stdin`) of the next, and so forth.

(a) Implement the pipe `|` operator:

`command1` `|` `command2` - The output of `command1` feeds the input of `command2`.

(b) Make sure your implementation works for an arbitrary number of commands:

`command1` `|` `command2` `|` . . . `|` `commandN`

Note that you are not required to implement piping for internal commands.