Assignment #1.2

CSD 2180 Operating Systems I: Man-Machine Interface

Deadline:	A1.2: As specified on the moodle
Topics covered:	Process creation and management
Deliverables:	To submit all relevant files that will implement the uShell program. Please upload the files to the moodle. Your program must be compile-able in the VPL environment using the `g++` compiler (std=c++17). A main function has been provided by VPL.
Objectives:	To learn and understand the basic of the internals of a shell program.

Programming Statement: A new shell program called uShell

In this assignment, you are tasked with the assignment to create a basic shell program called **uShell**. A shell program behaves typically in a similar fashion to a command prompt. Usually, it has the following capabilities:

- Ignore comments statement
- Perform internal commands
- Run programs and launch processes
- Setting of environment variables
- Perform input/output redirection (To be done later)
- Performing piping as a means of interprocess communications between processes (To be done in later)

1 An overview of uShell

Your program will be referred to as `uShell`. It is to be run either in interactive mode or as a shell script. Thus, the input to `uShell` can either be from the terminal or a file.

1.1 Usage

`uShell` is to be run as follows:

```
uShell [-v]
```

The bracketed arguments are optional. Their meaning are as the following:

- If the `-v` argument is used, the Shell is running in the verbose mode. In the verbose mode, every command line entered is repeated.

Some examples:

- `uShell` – No arguments at all. Input from terminal.
- `uShell -v` – Input from terminal, but every line entered is printed before execution.
- `uShell -v < test1.txt > result1.txt` – Input is taken from `test1.txt` and output is directed to `result1.txt`. Support input and output redirection.

1.2 Input

The program reads a command line one at a time from the standard input till a `exit` command or EOF from the file input is encountered. Every command line is terminated upon a newline character. Every command line is either empty or a sequence of *word*, where every *word* is a character sequence of printable characters but not whitespace characters[1]. The length of the command line is usually limited by the parameter ARG_MAX defined by Linux[2].

Hence, it is possible for a command line to be empty – it is simply a newline character or it is made up of only whitespaces. Furthermore, there is a special category of command line that can be ignored by the program – a comment. At any point in a command line, if the `#` sign is used, the rest of the line is treated as a comment and it can be ignored by the shell program. `uShell` does not consider quoting or backslash escaping.

1.3 Command line examples

The following shows some example of possible command lines. As you can see, there are both internal commands and external commands. The difference between the two will be made clear in later sections.

Command	Explanation
`# comment haha`	*This is a comment*
`/usr/bin/gcc`	*An external command - executing a program with full path given*
`ls -l`	*An external command - executing a program to be found among the directory list in environment variable PATH*
`echo hello`	*An internal command - printing the word hello to standard output*
`echo hello # silly comment`	*An internal command - printing the word hello to standard output.* `silly comment` *is ignored because it occurs after the* `#` *symbol*

When standard input is used as input, there should be a prompt printed before accepting input. For example,

```
uShell> echo haha # whatever
haha
uShell> ..
```

As explained in the next section, the default prompt is `uShell` but it can be changed by the `changeprompt` internal command.

[1] Whitespace is the set of blank characters, commonly defined as space, tab, newline and possibly carriage return

[2] You may use Linux command `getconf` ARG_MAX to get the value.

1.4 Internal Commands

An internal command is directly executed by the shell itself without executing another program. The first word in the command determines which internal is to be executed. The internal commands are:

- `echo`: print out the rest of the arguments.
- `exit <num>`: terminate the shell program with an exit value given by `num`. If `num` is not given, the default exit value is 0.
- `changeprompt <str>`: change the prompt to the string given by `str`.
- `setvar`: described in Section 1.5 on shell variables.

1.5 Shell Variables

Any word in a command line that begins with the character `$` and enclosed with curly braces refers to a shell variable. The default value of a variable is an empty string i.e., if a shell variable is used without any initialization, it is an empty string. The value can be changed via command line. The command to define and change the value of a variable is `setvar`. The usual syntax of the `setvar` command is as follows:

```
setvar <varname> <value>
```

The names of the shell variable are case-sensitive. To call out a shell variable, the '`$`' character is used to indicate that a variable is used. The following examples demonstrate the usage of the `setvar` command.

```
uShell> setvar HAHA hoohoo # assign the value hoohoo to HAHA
uShell> echo ${HAHA} # calling out the value of HAHA
hoohoo
uShell> # hoohoo is printed on the screen.
uShell> setvar haha # variable haha is defined and given a default value.
uShell> echo ${Haha}123 # Attempting to call out the value of an undefined variable.
Error: Haha is not a defined variable.
uShell> echo ${HAHA}123 # disambiguate the beginning and end of the var name
hoohoo123
uShell> echo ${HAHA }123 # wrong use of curly braces. Would be read as 2 separate words.
${HAHA }123
uShell> echo $${HAHA} # $ sign can be used together with variables
$hoohoo
uShell> echo ${${HAHA}} # nested use of curly braces are  not supported
${hoohoo}
uShell> # So the replacement only happen once.
```

Please ensure that your program prints out the same behaviour as given above.

1.6 changeprompt command

The examples of `changeprompt` are as follows:

```
uShell>changeprompt asd
asd>changeprompt gree sad
gree sad>changeprompt as_Asd    fd
as_Asd fd>changeprompt asfc # sad
asfc>changeprompt #asd
#asd>changeprompt asd asd#ds
asd asd#ds>exit
```

1.7 External Commands

An external command is a sequence of words where the first word does not match any of the internal commands given above. There are two forms of external command. They are:

- **Pathname commands**: where the first word will contain a '`/`' character. Example: `/usr/bin/gcc`. It must begin with a '`/`' character. We will only support what is called absolute pathnames.
- **Filename commands:** where the first word does not contain any '`/`' character. Example: `ls -l`.

For the external commands where the filenames only are given, the executable file is to be found in the list of directories given in the `PATH` variable. The `PATH` shell variable is a special variable that will be used in the context of the external commands. The `PATH` variable is a variable that is a sequence of pathnames separated by a colon character '`:`'. `PATH` variable is initialized from the PATH environment variable in Linux.

If the `PATH` variable is "`/bin:/usr/bin`", there are two directories given – `/bin` and `/usr/bin`. The shell program searches through the directories in the `PATH` variable to find the executable for running the external command. If the executable with the given filename is not found in all the directories or the given absolute path executable cannot be found, the shell program is supposed behave as the following example shows:

```
uShell> foo/goo/bin haha # not using an absolute path.
Error: foo/goo/bin cannot be found
uShell> /foo/goo/bin haha # example using non-existent path or filename
Error: /foo/goo/bin cannot be found
uShell> bin # bin cannot be found in any directories in PATH variable
Error: bin cannot be found
```

For the sake of ease, we ask that the user must provide the full filename and not merely filenames without the proper extensions. For example, when `haha` is not in the current directory, we have

```
uShell> ls haha
ls: cannot access 'haha': No such file or directory
```

If `haha` is in the current directory, we have

```
uShell> ls haha # haha is a file in current directory
haha
```

Normally, the external commands must be executed in a child process of the shell program and the shell program will immediately wait for the child process to finish before continuing execution.

2 uShell2.h

You will finish the following definition of member functions of the class `uShell2` in `uShell.cpp`, which is the only file required to submit. The class `uShell`, which is declared in Part 1, has been defined in `uShell_ref.cpp`. Its OBJ file is provided in VPL as `uShell_ref.obj` for linking.

```
#include <string>    //std::string
#include <vector>    //std::vector
#include <map>       //std::map
#include "uShell.h" //base

/*****************************************************************/
/*!
\brief
uShell2 class. Acts as a command prompt that takes in input and performs
commands based on the input.
*/
/*****************************************************************/
class uShell2 : public uShell
{
  protected:
  typedef void (uShell2::*fInternalCmd2)(TokenList const &);


  /*! Store the list of strings to function pointers of internal command
    in uShell2, i.e. for changeprompt and exit*/
  std::map<std::string, fInternalCmd2> m_internalCmdList2;

  /*****************************************************************/
  /*!
  \brief
  Reads the exit code from the token list
  \param tokenList
  Stores the exit code that will be extracted
  */
  /*****************************************************************/
  void exit(TokenList const & tokenList);

  /*****************************************************************/
  /*!
  \brief
  Changes the prompt that starts at every line
  \param tokenList
  The list of tokens where shell can retrieve the string value
  */
  /*****************************************************************/
  void changePrompt(TokenList const & tokenList);
```

```
/*****************************************************************/
/*!
\brief
Sets up the arugment lists required for calling the execvp function, and
creates the child process
\param tokenList
The list of tokens to get the data value from
\param startParam
The starting token to parse data from the list
\param endParam
1 pass the last token to parse data from the list
*/
/*****************************************************************/
void execute(TokenList const & tokenList,
                  unsigned startParam, unsigned endParam);


/*****************************************************************/
/*!
\brief
Calls an external command using the passed in parameters.
\param tokenList
The list of tokens to get the data value from
*/
/*****************************************************************/
void doExternalCmd(TokenList const & tokenList);

public:
/*****************************************************************/
/*!
\brief
Creates the class object of uShell2
\param bFlag
boolean value to decide whether to echo input
*/
/*****************************************************************/
uShell2(bool bFlag);

/*****************************************************************/
/*!
\brief
Public function for external call. Execute in loops and waits for input.
\return
Exit code, of the exit command
*/
/*****************************************************************/
int run();
```

};

2.1 void uShell2::exit(TokenList const & tokenList)

If there is no other argument, it just set `m_exitCode` to 0. Otherwise, it parses the input as `int` and set `m_exitCode` to the input value.

2.2 void uShell2::changePrompt(TokenList const & tokenList)

If `changeprompt` command is called but there is no input, the function just returns null value. Otherwise, update prompt.

2.3 void uShell2::execute(TokenList const & tokenList, unsigned startParam, unsigned endParam)

This function firstly checks whether the token[3] list is an empty argument list. It simply returns if the list is empty. Otherwise it gets the command line arguments and creates the argument array to put all the command line arguments. It then finds the executable file name and checks if this is an absolute path either started with '/' or './'. If so, it calls `execvp()` with the file name and the arguments. Else, it creates the directories list (e.g. type of $std :: vector < std :: string >$) from the environment variable `PATH`. Please note that the function can not be `const` because of the operator subscript of the map variable. It then loops through the directories list, and tries to invoke the input file name if it exists. At the end it deletes the created argument array. If no command has been found, it prints the error message.

2.4 void uShell2::doExternalCmd(TokenList const & tokenList)

The function flushes the buffer and then uses `fork()` to spawn child process. Child process will set the arguments to the input in the token list given to the process and call `execute()`. Then the child exits.The parent process should wait for the child to be done.

2.5 uShell2::uShell2(bool bFlag)

It sets function pointers for changeprompt and exit in $m\_internalCmdList2$.

2.6 int uShell2::run()

This function is enhanced based on the assignment A1.1. It has a loop and firstly check whether exit command is called. If so, it stops. Otherwise it prints out the prompt, with the right arrow. Then it gets user input. If there are no more lines from the input, it exits from the program (assume it gets re-directed input from the files). Otherwise, it clears the input buffer for next input. Obviously, it needs to skip if there is no input (e.g. empty line). It starts to tokenize the input otherwise. After this, it prints the input if verbose mode is set. It replaces all variables if possible. If the function call for replacement $replaceVars()$ returns false, an error has occurred, it continues to next line of input. Please note that replacement also clears comments, so we have to check if the result is empty. Next, it

[3] "Tokens" are smaller chunks of the string that are separated by a specified character, called the delimiting character.

finds if it is an internal command in the first list *m_internalCmdList*. If so, it activates the internal command. **Otherwise, it finds in the second list** *m_internalCmdList*2 **and activates the internal command if there is a match. If the search for the internal command fails, it regards the command as external command and does the external command accordingly.**[4] Please note that if the next char is EOF, we should end the loop to exit. Outside the loop, it returns exit code *m_exitCode*.

3 Rubrics

The total marks for this assignment is 100. The rubrics are as the following:

- `exit` command. Your `uShell` should be able to exit with the correct exit status value that is given from the input command line.
- `changeprompt` command. Your `uShell` should be able to change the prompt string when the user used the `changeprompt` command.
- External commands. You are expected to support both pathname and filename commands.
- Comments. Comments ought not to be verbose, but explain the big picture and use good variable names to indicate. Readability of your code is key here.
- Structure of your code. Avoid hard-coding.
- Pass the VPL test cases.
- The list above is non-exhaustive. The lecturer reserves the right to impose reasonable penalties for code that violates general practices or does not match the specification in an obvious way that has not been mentioned above. In exceptional cases, the lecturer reserves a discretionary right to allow resubmission or submission after the deadline.

[4]The only difference between run() in the parent class and that in the child class are the text in bold.