Assignment #1.3

CSD 2180 Operating Systems I: Man-Machine Interface

Deadline:	A1.3: 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 (A1.1 and A1.2)
- Perform internal commands (A1.1 and A1,2)
- Run programs and launch processes (A1.2)
- Setting of environment variables (A1.1 and A1.2)
- Perform background process (A1.3)
- Performing piping as a means of interprocess communications between processes (A1.3)

1 Introduction to A1.3

In A1.3, the student is expected to extend the same `uShell` program in A1.1 and A1.2. In particular, the student is expected to implement the following:

- An internal command called `finish`, to force the shell program to wait for background processes.
- Run background processes by supporting a syntax when a word "& " is appended to the end of the command line.
- Support multiple pipes between commands.

In this specification, we shall concentrate on describing these additional features. Please consult A1.1 and A1.2 for the other specifications.

2 Background commands and synchronization

The default behaviour of external commands is that they are executed before the next command is read. However, background jobs provide an opportunity for the user to launch jobs that the user does not have to wait for before running the next command. An example is `grep`, which is a command-line utility for searching plain-text data sets for lines that match a regular expression. Usually, one would need to finish searching before continuing, but background jobs give us a way to avoid that problem. The syntax for running a background job is by adding a "& " to the end of the command. When a background job is launched, the shell program responds by printing the process index (internal to the shell) and the process id (via the function `getpid()`). For example:

```
uShell> grep shell *.cpp & # grep will search cpp file for the string of shell
[0] process 1496
uShell> # grep is the first background job to launch, hence 0.
uShell> # grep's process id from getpid is 1496.
uShell> grep shell *.cpp& # wrong use of the &
grep: *.cpp&: No such file or directory
```

2.1 finish - Internal Command

The internal command that can be used in connection with this is the internal command `finish`. `finish` forces the shell program to wait for particular background processes to complete before continuing. The format for using `finish` is `finish` <process index>. The given process ID must be a valid one. For example:

```
uShell>gedit &
[0] process 10064
uShell>finish 0
process 10064 exited with exit status 0.
uShell>gedit &
[1] 12456
uShell>finish 0
Process Index 0 process 10064 is no longer a child process.
uShell>finish 1
process 12456 exited with exit status 0.
```

2.2 Pipes

Finally, an additional feature of `uShell` is the support of pipes. This is also a composite external command, where multiple external commands are executed concurrently. The syntax for pipes is the following:

```
ext_cmd1 | ext_cmd2 | ext_cmd3 | ...
```

where each `ext_cmd` is a sequence of words that form any legitimate external command as described above. The only difference is that these external commands cannot be background jobs (but you are not expected to perform error handling of this). You do not have to support piping with internal commands, although that may be a useful extension to implement.

Semantically, what the syntax mean is that the output of the preceding external commands is fed into the succeeding external commands. So the output of `ext_cmd1` is feeding into the input of `ext_cmd2` and so on and so forth. It is important to note that these commands are running concurrently, so there is no need to wait for `ext_cmd1` to complete before `ext_cmd2` runs. Finally, the last external command in the chain of pipes outputs to the stdout. Here are some examples of pipes:

```
uShell>ls
a config.obj my_setenv.c a.stackdump my_setenv.h
my_setenv.o main.cpp main.o Makefile
uShell>ls | grep setenv
my_setenv.c
my_setenv.h
my_setenv.o
uShell>ls | grep2 setenv | wc
Error: grep2 cannot be found
uShell>ls | grep setenv | wc
3 3 36
```

3 uShell3.h

You will finish the following definition of member functions of the class `uShell3` in `uShell.cpp`, which is the only file required to submit. The classes `uShell` and `uShell2`, which are declared in Part 1 and 2, has been defined in `uShell_ref.cpp` and `uShell2_ref.cpp`, respectively. The OBJ files are provided in VPL as `uShell_ref.obj` and `uShell2_ref.obj` for linking.

```
#include <string>    //std::string
#include <vector>    //std::vector
#include <map>       //std::map
#include "uShell2.h"

/*****************************************************************/
/*!
\brief
uShell3 class. Acts as a command prompt that takes in input and performs
commands based on the input.
*/
/*****************************************************************/
class uShell3 : public uShell2
{
  protected:
  typedef void (uShell3::*fInternalCmd3)(TokenList const &);


  /*! Store the list of strings to function pointers of internal command
      in uShell3, i.e. finish.*/
  std::map<std::string, fInternalCmd3> m_internalCmdList3;
```

```
/*****************************************************************/
/*!
\brief
Process info for background processes.
*/
/*****************************************************************/
struct ProcessInfo
{
  /*! The process ID given by OS */
  int PID;

  /*! The state of the process, whether it is active */
  bool bActive;

  /*****************************************************************/
  /*!
  \brief
  Default constructor
  */
  /*****************************************************************/
  ProcessInfo();

  /*****************************************************************/
  /*!
  \brief
  Value constructor, set process id and current state
  \param id
  The process ID given by the OS
  \param state
  The state of the process to set
  */
  /*****************************************************************/
  ProcessInfo(int id, bool state);
};

/*****************************************************************/
/*!
\brief
Process info for piping commands.
*/
/*****************************************************************/
struct PipeInfo
{
  /*! File descriptor array. The first is READ/IN descriptor, the
      second WRTIE/OUT descriptor */
  int descriptor[2];
```

```
  /*! The position of the pipe token "|" within the token list */
  unsigned posInToken;

  /*! Const value for READ/IN descriptor */
  static const int IN_DESCRIPTOR = 0;

  /*! Const value for WRITE/OUT descriptor */
  static const int OUT_DESCRIPTOR = 1;
};

/*! Store the list of background processes */
std::vector<ProcessInfo> m_bgProcessList;

/*!
\brief
Determine whether the command exists or not.
\param tokenList
The list of tokens to get the command and arguments from
\param startParam
The starting token to parse data from the list
\param endParam
The last token to parse data from the list
*/
/*****************************************************************/
bool exist (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);

/*****************************************************************/
/*!
\brief
Finish command: wait for a background process to finish
\param tokenList
The list of data to read in, the process ID to wait for
*/
/*****************************************************************/
void finish(TokenList const & tokenList);
```

```
  public:
  /*****************************************************************/
  /*!
  \brief
  Creates the class object of uShell3
  \param bFlag
  boolean value to decide whether to echo input
  */
  /*****************************************************************/
  uShell3(bool bFlag);

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

3.1 uShell3::ProcessInfo::ProcessInfo()

It sets `PID` to 0 and `bActive` to `false`.

3.2 uShell3::ProcessInfo::ProcessInfo(int id, bool state)

It sets `PID` to `id` and `bActive` to `state`.

3.3 void uShell3::finish(TokenList const & tokenList)

This function does `finish` command. It checks for `finish` command without parameters and indicates the error. If the `finish` command is with the correct parameters, it convert the argument to process id to wait for that process. It checks if the waited process ID is valid in the `m_bgProcessList` and also checks if the process to be waited for has already done by checking `bActive` for the process. If everything is OK, it waits and gets the returned status (using `waitpid()`). At the end it prints the status for the process and then set the process active flag to `false`.

3.4 bool uShell3::exist (TokenList const & tokenList, unsigned startParam, unsigned endParam)

This function determines whether the command exists or not. It gets the executable file name. If the file carries an absolute path, it checks whether the file exists. Otherwise, it sets up the environment variables list and loop through the list to find whether the file exists.

3.5 void uShell3::doExternalCmd(TokenList const & tokenList)

The function checks for rare case where first token is a pipe, where nothing will be done. It flushes the buffer and then count the number of pipes in the token list. It adds pipe information to a list typed of $std :: vector < PipeInfo >$ and creates the pipe (using `pipe()`. You may save the position of the pipe in the token list so that it can be used to separate the arguments. Once it has obtained the number of processes required, it loops over for further checking. It identifies where the beginning and ending position of the parameters in the token list. Otherwise if there is only a single process, it checks whether there is a background process token. It reports the syntax error if there is an empty string between the pipes. The function checks whether the command in the given token list with the specified starting and ending position of the parameters exists by calling `exist()` and reports the error accordingly. It returns if there is an error.

Otherwise, it starts to create the child processes using `fork()`. For each `fork()`, if background process is specified, the parent does not wait and adds the child to the list by pushing into `m_bgProcessList`. Please note you need to print out the status for the child process. Otherwise if no background process is required, the parent just adds the child PID to the list of the processes with piping. For the child process, it sets the arguments to the input in the token list given to the process and call `execute()`. If the process is not the first one to be created, it closes stdin and replaces with pipe in. If the process is not the last one, it closes stdout and replaces with pipe out. It closes all pipes that are not needed anymore when calling `execute()` given in `uShell2.h`. Then the child exits by calling `_exit(EXIT_FAILURE)`. The parent process should close all pipes before waiting for all the child processes to be done.

3.6 uShell3::uShell3(bool bFlag)

It sets function pointers for `finish` in $m\_internalCmdList3$.

3.7 int uShell3::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 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 $m\_internalCmdList2$ and activates the internal command if there is a match. **Else, it continues to search the internal command in the third list $m\_internalCmdList3$. If the search for the internal command fails, it regards the command as external command and does the external command accordingly.**[1] Please note that if the next char is EOF, we should end the loop to exit. Outside the loop, it returns exit code $m\_exitCode$.

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

4 Rubrics

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

- Your background jobs are launched properly and finish works as expected, including the error messages that are described above.
- Pipe commands should work as expected for any number of external 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.