CREATING YOUR OWN SHELL

OVERVIEW

This assignment will require you to implement a simplified shell that parses a command entered by the user, creates new processes to execute specific programs, and performs input and output redirection and background execution. Your shell must prevent the creation of zombie processes by handling process termination appropriately. Your shell will not contain all the functionality of a production shell, but will be sufficient to demonstrate the basics in process creation, destruction, and synchronization.

THE PROGRAM

This is a two-part project to implement the program `myshell`. In the first part, your program must parse an input string into a structure that can be used by the second part to create a new process, handle input and output redirection and background/foreground execution.

PART 1:

Your program must perform the following actions:

1. Display a prompt on *stdout*.
2. Accept a command as a string from the user (input string will terminate with a newline character). The program must terminate when the command *exit* is entered.
3. Parse the input string into tokens, which are single words delimited by one or more spaces, tabs, or newline characters.
4. Store the tokens in the fields of class *Param*.
5. Print the contents of the fields using `printParams()` function, which is provided and explained below, but only when the shell is started with the debug option *–Debug*. Otherwise, the contents of the string will not be printed.
6. Return back to step 1.

The class described below is used to store the parsed input. I have included a defined value to indicate the maximum number of tokens you will find on the command line.

```
/* don't test program with more than this many tokens for input */
#define MAXARGS 32
/* class to hold input data                                                  */
/* feel free to add missing constructor, destcructor, and getter/setter functions */
class Param
{

  private:
     char *inputRedirect;           /* file name or NULL                  */
     char *outputRedirect;          /* file name or NULL                  */
     int  background;               /* either 0 (false) or 1 (true)       */
     int  argumentCount;            /* number of tokens in argument vector */
     char *argumentVector[MAXARGS]; /* array of strings                   */
```

```
  puplic:

   void printParams()
   {
       cout << "InputRedirect: [" <<
               (inputRedirect != NULL) ? inputRedirect : "NULL" << "]" << endl <<
               "OutputRedirect: [" <<
               (outputRedirect != NULL) ? outputRedirect : "NULL" << "]" << endl <<
               "Background: [" << background << "]" << endl <<
               "ArgumentCount: [" << argumentCount << "]" << endl;
       for (int i = 0; i < argumentCount; i++)
           cout << "ArgumentVector[" << i << "]: [" <<
                   argumentVector[i] << "]" << endl;
   }

};
```

Notice that the first four class variables are special. The first two indicate that either input redirection or output redirection is desired. The third indicates whether the background operator has been found on the command-line. The fourth indicates the number of tokens stored in the argument vector. Consider this line of input shown with the prompt *$$$*.

```
$$$ one two three <four >five &
```

When the line is parsed, the first three tokens are not special, so they must be placed in `argumentVector[0]`, `argumentVector[1]`, and `argumentVector[2]` accordingly. When the fourth token is extracted, it is identified as an input redirection because of the beginning character ('<'). The characters following immediately the redirection indicator form the name of the file from which input must be redirected. The name of the input file ("four") must be stored in `inputRedirect`. Similarly, the beginning character ('>') of the fifth token identifies output redirection and the characters following the redirect character specify the name of the file to which output must be written. The name of the output file ("five") must be stored in `outputRedirect`.

You may or may not allow for spaces following the input and output redirection indicator. If you allow for spaces, the token following the redirection indicator must identify the filename. If you disallow spaces and no filename is specified, an error message must be generated. Backgrounding is indicated by the ampersand (&) and is recorded by setting the value of `background` to 1. If it is included in an input string, it must always appear as the last character in the string. Overall, an acceptable input is a single text line ending by a new line character that follows the syntax as shown below:

```
[token [' '|'\t']+]* [token] [' '|'\t']+
    [<input [' '|'\t']+] [>output [' '| '\t']+ ] [&]
```

In this notation, `[ ]` indicates an optional parameter, `*` indicates 0 or more times the value in the brackets, and `|` indicates alternatives.

Once the input line is parsed and the *Param* object variables are properly set, you must print the object's data with the `printParams()` function but only when the shell is in debug mode.

PART II:

You must extend the previous program to create a basic shell program. The functionality of your shell program is similar to that of a production shell such as a c-shell or a bourne shell found in many operating systems. Input from a command line is read, interpreted, and executed. Input/Output redirection as well as foreground/background

execution must also be implemented. However, interprocess-communication will not need to be implemented for this project.

Your program is not required to be "bulletproof". You can assume that input data will follow the description given in class. You may NOT assume that input data is valid. That is, the command may be misspelled. For example, I could test your code by entering the command *cqt file* instead of *cat file*. Your shell must provide a short and descriptive message explaining the error that has occurred. Your shell must continue processing commands until the user enters the exit command. Before terminating execution, your shell must make certain that all background child processes have exited. In other words, the parent cannot terminate until all children have terminated.

For testing your shell, you need to download the *slow* program from *eLearning* in the Content area for Processes. The slow process will allow you to test if you handled the execution of background processes properly.

Your shell must be started from the command line according to the following syntax:

```
myshell [-Debug]
```

The option –Debug may be provided to print out the structure of a parsed command. Be sure that your Makefile produces the program `myshell` and not any program name of your choice.

SAMPLE TEST CASES

The following describes a list of test cases as a starting point for testing your work. Your program must pass at least these tests for you to receive full credit. Additional test cases may be used during grading.

Command	Explanation
`ls –l`	shows a listing of files in the current directory
`ls –l >testfile.txt`	writes a listing of files into the text file *testfile.txt*
`grep –i shell testfile.txt`	list many lines containing the word shell in the previous file
`cat <myshell.c`	displays the source code of the program on the screen
`cat <myshell.c &`	as above except the output will be displayed in the background causing the prompt of the shell to be mixed with the output of the file
`cat testfile.txt &`	displays the content of the text file *testfile.txt* on the screen in the background
`./slow &`	runs the program slow from the current working directory in the background
`exit`	terminates the shell; all child processes must be terminated for the shell to close avoiding creation of zombie processes

SUGGESTED SYSTEM CALLS

Most implementation details are left for you to decide. Below is a list of system calls that you must use in the implementation of your program. You may use additional system calls as necessary. You may not use the `system(3)` function, nor use assistance from a production shell program to do any of the work for you.

For part I:

- `fgets(3)` – input characters and strings
- `strtok(3)` – extract tokens from strings
- `printf(3)` – formatted output conversion

For part II:

- `exec(2)` – (`execl()` or `execlp()` or `execv()`, …) execute a file
- `fork(2)` – create a new process (exactly like the parent process)
- `freopen(3C)` – opens a stream (change an open stream to some other file)
- `waitpid(2)` or `wait(2)` – wait for a child process to change state (or terminate)

DELIVERABLES

Your project submission must follow the instructions below. Any submissions that do not follow the stated requirements will not be graded.

1. Follow the submission requirements of your instructor as published on *eLearning* under the Content area.
2. You must have at a minimum the following files for this assignment:
    a. myshell.cpp (implements the shell)
    b. parse.cpp (implements the command line parser of the shell)
    c. parse.h (defines the functionality you want to expose)
    d. Makefile
    e. README

The README file describes purpose and usage of your program. The file `parse.cpp` provides the parse functionality implemented in *Part I* to parse the input string into the given structure. It is important that you refactor your code so that the main function is in `myshell.cpp`. Keep in mind that documentation of source code is an essential part of programming. If you do not include comments in your source code, points will be deducted.

TESTING & EVALUATION

Your program will be evaluated on the Department's public Linux servers according to the steps shown below. Notice that warnings and errors are not permitted and will make grading quick!

1. Program compilation with Makefile. The options *–g* and *–Wall* must be enabled in the Makefile. See the sample Makefile that I uploaded in *eLearning*.
    - If errors occur during compilation, there will be a substantial deduction. The instructor will not fix your code to get it to compile.
    - If warnings occur during compilation, there will be a deduction. The instructor will test your code.
2. Perform several evaluation runs with input of the grader's own choosing. At a minimum, the test runs address the following questions.
    - Are commands created correctly using `fork()` and `exec()`?
    - Is input/output redirection correctly implemented?
    - Is foreground/background processing correctly implemented?
    - Does the program terminate correctly when exit is entered?
    - Does the shell program wait for all children to terminate before it terminates?

DUE DATE

The project is due as indicated by the Dropbox for project 1 in *eLearning*. Upload your complete solution to the Dropbox. I will not accept submissions emailed to the grader or me. Upload ahead of time, as last minute uploads may fail.

GRADING

This project is worth 100 points in total. The rubric used for grading is included below. Keep in mind that there will be deductions if your code does not compile, has memory leaks, or is otherwise, poorly documented or organized.

Correct Submission Format	**Perfect**	**Deficient**		
Canvas	5 points individual files have been uploaded	0 points files are missing		
Compilation & Documentation	**Perfect**	**Good**	**Attempted**	**Deficient**
Makefile	5 points make file works; includes clean rule	3 points missing clean rule	2 points missing rules; doesn't compile project	0 points make file is missing
Compiles	10 points no errors, no warnings	7 points some warnings	3 points many warnings	0 points errors
Documentation & Style	**Perfect**	**Good**	**Attempted**	**Deficient**
documentation & program structure	5 points follows documentation and code structure guidelines	3 points follows mostly documentation and code structure guidelines; minor deviations	2 points some documentation and/or code structure lacks consistency	0 points missing or insufficient documentation and/or code structure is poor; review sample code and guidelines
Shell	**Perfect**	**Good**	**Attempted**	**Deficient**
tokenization	10 points correct, completed	7 points minor errors	3 points incomplete	0 points missing or does not compile
parameter processing	10 points correct, completed	7 points minor errors	3 points incomplete	0 points missing or does not compile
creates new process	15 points correct, completed	11 points minor errors	4 points incomplete	0 points missing or does not compile
input & output redirection	15 points correct, completed	11 points minor errors	4 points incomplete	0 points missing or does not compile
backgrounding	10 points correct, completed	7 points minor errors	3 points incomplete	0 points missing or does not compile
avoids zombies	15 points correct, completed	11 points minor errors	4 points incomplete	0 points missing or does not compile

I will evaluate your solution as attempted or insufficient if your code does not compile. This means, if you submit your solution according to my instructions, document and structure your code properly, provide a makefile but the submit code does not compile or crashes immediately you can expect at most 25 out of 100 points. So be sure your code compiles and executes.

COMMENTS

I strongly recommend you starting to work on this project right away to leave enough time for handling unexpected problems. Insert many output statements to help debug your program. You should consider using debugging techniques from the first programming practice I posted in *eLearning*. Compile with the `-DDEBUG=1`

option until you are confident your program works as expected. Then recompile without using the `-D` option to "turn off" the debugging output statements. Note the `DEBUG` option provided to the compiler is different from the `DEBUG` option for program execution. The compiler uses `-DDEBUG=1` to compile your code with any debug information you provided via preprocessor directives.