DISTRIBUTED COMPUTING

OVERVIEW

This assignment will require you to implement a new shell that parses a command entered by the user and creates a set of new processes to run a distributed computing application. The shell requires redirection of input and output and prevention of zombie processes. Your shell will not behave like a typical shell to process commands at the command line; rather it is designed to help users launch multiple processes simultaneously to solve computing problems cooperatively.

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. The structure must then be used by the second part to 1) create a set of new processes based on the input on the shell's command line and 2) handle input and output redirection for the new processes.

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 a provided structure shown below.
5. Print the contents of the structure using the `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 structure used to store the parsed input is shown below. 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
/* structure to hold input data */
struct PARAM
{
  char *inputRedirect;           /* file name or NULL */
  char *outputRedirect;          /* file name or NULL */
  int  argumentCount;            /* number of tokens in argument vector */
  char *argumentVector[MAXARGS]; /* array of strings */
};

/* a typedef so we don't need to use "struct PARAM" all the time */
typedef struct PARAM Param_t;
```

Notice that the first three components of the struct are special. The first two describe the file name for either input redirection or output redirection, if desired. The third describes the number of tokens in the argument vector entered at the command-line. 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 because they do not start with a beginning character such as '<' or '>', so they should be placed in `argumentVector[0]`, `argumentVector[1]`, and `argumentVector[2]` respectively. The argument counter named `argumentCount` should be set to three. 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 should be read. The name of the input file ("four") should be stored in `inputRedirect`. Similarly, the beginning character ('>') of the fifth token identifies output redirection and the characters following the redirect character specifies the name of the file to which output should be sent. The name of the output file ("four") should be stored in `outputRedirect`. Please do not allow for spaces between the beginning character for redirection ('<' or '>') and the file name. If input or output redirection is not specified on the command line, your program needs to set the corresponding fields in the structure to NULL. 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, `+` indicates 1 or more times the value in the brackets, and `|` indicates alternatives.

Once the input line is parsed and the structure elements are properly set, you must print the structure with the following function when the shell is in debug mode.

```
void printParams(Param_t * param)
{
       int i;
       printf ("InputRedirect: [%s]\n",
         (param->inputRedirect != NULL) ? param->inputRedirect:"NULL");
       printf ("OutputRedirect: [%s]\n",
         (param->outputRedirect != NULL) ? param->outputRedirect:"NULL");
       printf ("ArgumentCount: [%d]\n", param->argumentCount);
       for (i = 0; i < param->argumentCount; i++)
             printf("ArgumentVector[%2d]: [%s]\n", i, param->argumentVector[i]);
}
```

PART II:

You must extend the previous program to create a specific shell program that runs distributed computing solutions. The functionality of your shell program is different from that of a production shell such as a c-shell or a bourne shell found in many operating systems because the shell will launch several instances of a program. Your shell reads input from a command line and interprets it. Your shell must handle input/output redirection and interpret an argument vector to run a specified program with its own parameter as follows:

- `argumentVector[0]`: this element will define the name of a program that must be executed.
- `argumentVector[1]`: this element will define a counter that specifies the number of times the program must be launched as a separate process/instance.

- All other elements in the argument vector must be passed to the specified program.

Upon interpreting the values in the argument vector, your shell must launch *k* instances of the specified program using system call `fork()` where *k* is the number specified in `argumentVector[1]`. It must generate a new argument vector for each process to become the command line arguments read in `main()`. However, in order for the processes to work cooperatively on solving a problem the shell must generate a separate index for each process and passes it to the process in the new argument vector. The program will use this index to apply a divide and conquer strategy on the data for solving a problem. The new index should be inserted as a separate value into the previously generated argument vector to create a new argument vector. You must decide for yourself whether you can use the existing argument vector in the structure or create a new argument vector to store the new values.

Here is an example of how the shell should work assuming that $$$ is the prompt:

```
$$$ ./collatz 4 10000
```

In this example, the shell must launch a local program called collatz 4 times. It provides each collatz process the command line arguments that include the following:

```
collatz 4 i 10000
```

Here, *i* is the index of the instance that the shell forks. It should range from 0 to 3 for the corresponding instance of collatz. Each instance of collatz may use i and 4 to decide what range of numbers among the specified 10,000 it must process.

Your shell must handle the following error cases. Users may enter an incorrect program name referring to a program that does not exist on your file system or enter a character string as the counter value that is not an integer, or enter nothing. Your job should be to alert the user if the program you attempt to execute does not exist or if the second argument is not an integer. Take advantage of the appropriate system calls mentioned below to determine if a program exists or not. Furthermore, your shell must handle empty inputs gracefully and not crash.

Your shell must use system call `waitpid()` to wait for all the processes that you launched to terminate before accepting the next command. Finally, your shell must exit when you enter exit. For testing your shell, you may use the test programs `testme` and `prime` located on the shared drive under:

```
/shared/dropbox/treichherzer/cop4635/resources.
```

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
`testme 5 1000 >output.txt`	launches 5 instances of *testme* and redirects output to output.txt
`collatz 8 <input.txt >output.txt`	launches 8 instances of collatz and redirects input and output; here each collatz instance only receives 8 and the index of the instance as part of the argument list in main()
`exit`	terminates the shell

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 should 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 should have at a minimum the following files for this assignment:
    a. myshell.c (implements the shell)
    b. parse.c (implements the command line parser of the shell)
    c. parse.h (defines the functionality you want to expose)
    d. Makefile
    e. README (optional if project wasn't completed)

The file `parse.c` 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.c`. 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.

Your program will be evaluated according to the steps shown below. Notice that I will not fix your syntax errors. However, they 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?
    - Does the shell launch the correct number of processes?
    - Does the shell avoid zombie processes?
    - Does the shell terminate correctly when exit is entered?

DUE DATE

The project is due as indicated by the Dropbox for project 1 in *eLearning*. Upload your complete solution to the dropbox and the shared drive. I will not accept submissions emailed to me or the grader. 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.

Project Submission	**Perfect**	**Deficient**		
eLearning	5 points individual files have been uploaded	0 points files are missing		
shared drive	5 points individual files have been uploaded	0 points files are missing		
Compilation	**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
compilation	10 points no errrors	7 points some warnings	3 points some errors	0 points many errors
Documentation & Program Structure	**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
myshell	**Perfect**	**Good**	**Attempted**	**Deficient**
tokenization	15 points correct, completed	11 points minor errors	4 points incomplete	0 points missing
parameter processing	10 points correct, completed	7 points minor errors	3 points incomplete	0 points missing
creates new process	15 points correct, completed	11 points minor errors	4 points incomplete	0 points missing
input & output redirection	15 points correct, completed	11 points minor errors	4 points incomplete	0 points missing
avoids zombies	15 points correct, completed	11 points minor errors	4 points incomplete	0 points missing

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 23 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 lots of 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.