CSS 543
Program 1: Shell-Like Command-Line Interpreter

Professor: Munehiro Fukuda

Due date: see the syllabus

1. Purpose

This assignment implements a simple command-line interpreter that allows fore/background process execution, stdin/stdout redirection, and stdout piped to stdin as seen in the Unix shell. Through the implementation, you will exercise the use of several system calls such as: fork, exec(vp), wait(pid), open, dup, read, write, and close.

2. Interpreter Specification

The command-line interpreter that you will implement should have the following features:

(1) *Prompt:* the interpreter prints out a prompt sign like "$" or ">" every time a user types the "return" key.

(2) *Foreground execution:* the interpreter allows a user to run a program as a new process and waits for the termination of this process.

```
$ command
```

(3) *Sequential execution of multiple processes:* if a user types multiple commands or programs, each delimited with ";", the interpreter run these commands or programs one by one in sequential. The last command/program does not need a ";" delimiter unless it needs to be put in the background with "&".

```
$ commandA; commandB; commandC
```

(4) *Background execution of single or multiple processes:* if a user types one or more commands or programs, each delimited with "&", the interpreter run these commands or programs in parallel. It does not have to wait for the termination of any of these processes.

```
$ commandA& commandB & commandC&
```

(5) *Mix of sequential and background execution:* the interpreter allows a user to type multiple commands or programs, each delimited with ";" or "&". Needless to say, the interpreter must wait for the termination of a process which was delimited with ";", while proceeding to the next command/program without waiting for the current process termination if it was delimited with "&".

```
$ commandA& commandB ; commandC& commandD
```

(6) *Standard input/output redirection:* "< filename" redirects a given command/program's stdin to filename. "> filename" redirects a given command/program's stdout to filename.

```
$ commandA < input_file > output_file
```

(7) *Standard output piped to standard input:* "A | B" means that process A's standard output is piped to process B's standard input. The interpreter allows multiple pipes such as "A | B | C | D | E".

```
$ commandA | commandB | commandC | commandD
```

(8) *Mix of the above:* the interpreter allows any "possible" combination of all the above features:

```
$ commandA < input_file | commandB | commandC > output_file & commandD; commandE&
```

3. Statement of Work

Implement the command-line interpreter specified above, using Unix system calls. As far as you use Unix system calls, you may use any platform such as Unix, AIX, Linux, Mac OS, Open Solaris, and Cygwin. Needless to say, please use C or C++ as your programming language. Since lexical analysis is not our main focus, you may use the following template if you want. The same code is also available as uw1-320-lab.uwb.edu: ~css543/programming/project1/shell_template.cpp.

```
#include <iostream>
#include <vector>
#include <stack>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>
#define MAX 256
#define STDIN 0
#define STDOUT 1
using namespace std;

void extractCommand( char buffer[], int top, int cur, char delim,
                     vector<char**> &commands, vector<char> &delims ) {
  // check if there is a command between top and cur
  if ( top == -1 || top == cur )
    return;

  // prepare a space for this command
  char *command = new char[MAX];
  bzero( command, MAX );

  // copy the command from buffer
  bool isDelim = ( delim == ';' || delim == '&' || delim == '|' );
  if ( isDelim )
    strncpy( command, buffer + top, cur - top );     // remove the delimitor
  else
    strncpy( command, buffer + top, cur - top + 1 ); // include the last char

  // tokenize all argunents of the command string.
  char *sep = " \t";   // tokens
  char **arguments = new char*[MAX];
  int index = 0;
  for ( char *arg = strtok( command, sep ); arg != NULL; arg = strtok( NULL, sep ) ) {
    arguments[index++] = arg;
  }
  arguments[index] = NULL;

  // register command/arguments to the command list
  commands.push_back( arguments );
  if ( isDelim )
    delims.push_back( delim );
  else
    delims.push_back( ';' );
}

int main( int argc, char *argv[] ) {

  char buffer[MAX];              // a line buffer

  // keep reading line by line
  while ( cin ) {
    // print out a command prompt
    cout << "% ";

    // reinitialize and read a command line
    bzero( buffer, MAX );
    cin.getline( buffer, MAX );

    // delimit each command with |, ;, and &
    vector<char**> commands;  // a list of commands/arguments
    vector<char> delims;      // a list of delimitors such as | ; &
    int top = -1;             // the current command top in buffer
    for ( int i = 0; i < strlen( buffer ); i++ ) {
      // skip unnecessary spaces
      if ( top == -1 && ( buffer[i] == ' ' || buffer[i] == '\t' ) )
        continue;
```

```
      // set the next command top
      if ( top == -1 )
        top = i;


      // reach the end of the current command
      if ( i == strlen( buffer ) - 1 ||
           buffer[i] == ';' || buffer[i] == '&' || buffer[i] == '|' ) {

        // extract current command from buffer
        extractCommand( buffer, top, i, buffer[i], commands, delims );
        top = -1;
      }
    }

    // The following blue code shows all commands in commands as well as
    // their corresponding delimiters in delims
    vector<char**>::const_iterator ci;
    // for debugging
    int i = 0;
    for ( ci = commands.begin( ); ci != commands.end( ); ci++ ) {
      char **arguments = *ci;
      for ( int j = 0; arguments[j] != NULL; j++ )
        cout << "command[" << i << "][" << j << "]: " << arguments[j] << endl;
      i++;
    }

    vector<char>::const_iterator di;
    // for debugging
    i = 0;
    for ( di = delims.begin( ); di != delims.end( ); di++ )
      cout << "delimitor[" << i++ << "]: " << *di << endl;

    // You should add your code here to interrupt each command that is stored in commands.

    commands.clear( );
    delims.clear( );
  }
}
```

Note that the above template code lexically analyzes a standard input line, stores each command and its arguments in char[][], (i.e., char[0][] is a command; char[1][] is 1st argument; char[2][] is 2nd argument; char[n][] is nth argument; and char[n+1][] = NULL), stores all commands in the commands vector<char**>, and stores all the corresponding delimiters, (i.e., ';' or '&') in the delims vector<char>. All you need is, "as seen in the above blue code", to retrieve each command from the commands vector as well as the corresponding delimiter from the delims vector and to execute each command with child/grand/great-grand child processes.

4. What to Turn in

This programming assignment is due at the beginning of class on the due date. Please turn in the following materials in a hard copy. No email submission is accepted.

Criteria	Grade
Documentation of your algorithm including explanations and illustration in one or two pages. Insufficient or too-much (i.e., less than 1 or more than 2 pages) documentation receives 4pts.	5pts
Source code that adheres good modularization, coding style, and an appropriate amount of commends. 5pts: well-organized and correct code receives 4pts: messy yet working code or code with a minor error 3pts: code with two errors/bug 2pts: code with 3+ errors/bugs or incomplete code.	5pts

Execution output that verifies the correctness of all your implementation as well as covers many test cases. 5pts: a correct output with full test cases 4pts: an output with a minor error or a little insufficient test cases (missing a test for one of the features) 3pts: an output with two errors/bug or a little insufficient test cases (missing tests for two of the features) 2pts: an output with 3+ errors/bugs or quite incomplete test cases (missing tests for 3+ of the features)	5pts
Discussions about the limitation of your program and possible functional improvement in one page.	5pts
Extra Credit: Clearly mention if you even added more features to the original specification (such as >&, >2, ( ), shell variables, etc.), in which case 1 extra credit will be considered. Note that implementing "exit" or "quit" is too easy to count as an extra credit. Also, an extra credit is counted only when you received 5pts for your source code and execution output respectively.	(1pts)
Lab Sessions 1 through 2 counts 1pt for each. If you have not yet turned in a hard copy of your source code and output or missed any session(s), please turn in together with program 1.	2pts
Total Note that program 1 takes 15% of your final grade.	22pts