EECE 315
Project/Assignment #2

This is a **group assignment**. All group members must equally contribute, cooperate and coordinate.

The Part 1 of this assignment also includes activities that are to be demoed in the lab to the TAs.

==

PART 1 (In-lab Activities)

==

The following activities are to be demoed to the TA in the lab on the week of Jan 20 to 23 during your own lab section. Attempt the activities before going to the lab to expedite the demos.

For activities 1 and 2, see the videos on Connect for the basics of make, gdb and Valgrind.

Activity 1) gcc and make: Two very common and useful programming tools for the UNIX-based systems are *gcc* (GNU Compiler Collection: http://gcc.gnu.org/), and *make* (GNU Make: http://www.gnu.org/s/make/).

A) Assume that you have a C source code file named *myProgram.c* (the whole program written in one file) and assume that there is no dependency. What is the *gcc* command that will compile this file and build it to an executable called *myProgramExe* when issued in the UNIX/Linux command line? What is the default executable if you do not specify the output filename?

B) Note that the executable file name in the first part of part A has no extension. Is that ok? Also explain how you would run an executable from the command line.

C) Now use the *make* utility. Create a *makefile* that can be used to compile the file (with a result exactly similar to part A). Explain how you call *make* from the command line for this task (Note: you will use a quite similar makefile for the programming question).

D) Are UNIX file (or folder) names case-sensitive or case-insensitive?

Deliverable: Confirm with your TA in the lab that this task is done.

Activity 2) gdb and Valgrind: Two useful and powerful debugging tools for the UNIX-based systems are *gdb* (GNU Project Debugger: https://www.gnu.org/software/gdb/), and *Valgrind* (http://valgrind.org/).

A) Install Valgrind on your Ubuntu machine. Follow the step provided under quick start: http://valgrind.org/docs/manual/quick-start.html#quick-start.prepare

B) Explain how you should compile your C code using gcc to be able to use gdb.

C) Explain the procedure and command you need to you can use gdb to debug your code by stepping though each statement. Also explain how you set and remove breakpoints.

Deliverable: Confirm with your TA in the lab that this task is done.

Activity 3) Linux Module: Follow the instructions on page 96 to 101 of your textbook (Linux Kernel Module Programming Project) to write and add a simple module to your Linux on Virtualbox.

Note that we will use this exercise to practice using git and github as well.

Deliverable: Confirm with your TA in the lab that this task is done.

Activity 4) Practice with git and github: Follow the TA's instructions to practice using git and github using the code from activity 3.

See the info and tutorial on git and github on Connect (under "Readings/git and github/")

Deliverable: Confirm with your TA in the lab that this task is done.

==

PART 2 (Programming)

==

Deliverable: Submit **to Connect** one **zip file** that includes the final version of all your files (for the programming question and the project). Please find the guidelines for the submission and on the project report below.

Any one of your group members may submit for the group (i.e. all group members will see the same submission dropbox), please coordinate. You may submit as up to two times up to the deadline. We will consider the last submission for marking.

Enclosure: An info file related to this assignment on an introduction to "Executing Commands in UNIX" has also been posted.

Due date: Friday January 30, 2015 by 3:00 PM; This is for Part2, all in-lab activities are to be done in the lab as described before.

==

Github: you will be using github to make it easier for you to collaborate and write the programming exercises of part 2 as a team.

In addition, **every group member** is also required to record at least a snapshot (using commit and push) to gitthub once on January 21st, once on January 26th, and once when you make your final contribution to the project (if you make any changes after the above second snapshot). Create and use the folder PA2 (locally and on github) to hold the files for this assignment.

Submission to Connect Guideline**:** Submit one .zip file that includes one .pdf file and two directories.

- The .pdf file is your project report (see below).
- One directory includes the c file(s) for the Win32 programming question.
- The other directory includes the project files (simple shell). You must include the makefile and all dependencies.

- Please only use the formats specified. That is .zip for the main file (should easily be opened in Ubuntu or Windows), .pdf (or html, to be portable, there are many free pdf creator software), and the .c file formats. Please name the zip file `GroupNum.zip` where `GroupNum` is *your group number*.
- All .c codes should include sufficient comments, and the project should compile and work correctly according to the specification.
- For better code readability, please set the C code width to the standard 80 characters per line in your programs.
- Some marks will be dedicated to following the guidelines.

Project report: The project is explained in a formal report that has the following sections:

Front page - names and student numbers of all members in your group, title, date sunmitted, ...

Contribution page - Clearly state the contribution of each group member in this project. Mention the contributions both in percentage and the contributed components.

This info page should be in percentage, as well as in words specifying which tasks, design, coding, debugging, etc each group member contributed to.

Introduction - A brief description of the introductory and required information related to the project, such as the problem statement, high-level flowchart and the project benefits.

Solution - A complete description of your approach to solving the problem, such as the implementation, techniques, and explanation of your code (the code itself is submitted by the .c files; see submission guidelines).

Debugging - An outline of any problems you ran into, any test sets used to check the correctness of the code, ...

The report should include relevant figures, comments, test cases, conclusion, and any appendixes needed. Do not include the complete code with the report, though including small code segments that are necessary for your explanations is fine.

The length of the report is up to you. It should be just long enough to sufficiently explain the project as outlined above. Use an 11pt font size (Times New Roman). It should not be longer than 4 pages (excluding the front page, contribution page and any possible appendixes).

You may be invited later on to defend/present your project.

===

Programming Problem – Problem 4.26 of the textbook (you only need to write the program using the Win32 thread library)

==

Project - A Simple Shell

This exercise may be solved on any UNIX system. For consistency, implement your project on an Ubuntu Linux distribution only. The project is taken verbatim (with additions and slight modifications) from the operating systems book by G. Nutt. As usual all code must be in C.

An introductory skeleton code is provided which serves as a starting point/suggestion.

Primary purpose: Your shell must execute Ubuntu's external commands that the user types at the prompt. External commands are those commands that are stored as separate executable files/binaries. Most Linux/UNIX command-line commands are external.

In your solution you are required to use *execv()* instead of *execvp()*, which means that you will have to read the *PATH* environment variable, then search each directory in the *PATH* for the command file name that appears on the command line.

Feature 1 - Internal commands: Some shell commands are internal, that is, they are shell built-in commands. Examples of such commands are cd, pwd, and echo. You do not need to "implement" internal command internally in the shell, though you must consider an alternative way so that your shell executes the two important internal commands of cd and pwd (Hint: for example, use *system()*.) The shell should also respond to the *exit* and *quit* commands.

Feature 2 - Running in the background: Similar to an Ubuntu shell, your shell must be able to run a command in the background. Implement the '&' modifier so that if the last character on the command line is '&', the program is executed in parallel with the shell, rather than the shell's having to wait for it.

Feature 3 - Implement redirection: normally the standard input or output file are stdin and stdout. Allow that the standard input or output files to be redirected using the '>' symbol. (Hint: this feature is more involved than the previous two. Add it after you have a working shell). Also Note that the implementation of '<' redirection is not a requirement here.

Introduction

Start with reading the attached introductory document ("executing commands in UNIX"). Then starting with the code explained in the that document, design and implement a simple, interactive shell program that prompts the user for a command, parses the command, and then executes it with a child process.

Background

The OS exports its functionality as the system call interface. All non-OS software uses OS services by making function calls such as *read()* and *fork()* on the system call interface. Human users such as a batch computer operator or an interactive user also need to interact with the OS so they can run programs, inspect the collection of files, and so on. Should the OS provide a specialized human-computer interface just for this purpose? In modern computer systems, the OS does *not* include such an interface. Instead, there are one or more command line interpreter programs that use the conventional system call interface to invoke OS services, and which export an "operator's console" to a user. Since a command line interpreter is just an application program, programmers can create their own if they don't like the ones delivered with the OS.

The original UNIX developers were the first to adopt this technique for constructing a command line interpreter. They called their command line interpreter the **shell program**, a name that has stuck (and is now used to refer to any program that provides a human-OS interface). The inspiration for the name is that the shell program provides a protective cover over the OS, much like a shell protects an oyster.

The simplest shell programs are text-based programs (more complex shell programs use graphics with point-and-select interfaces). A text-based shell assumes a screen display with a fixed number of lines (usually 25) and a fixed number of characters (usually 80) per line. The user interacts with the OS by typing a string of characters (terminated with the "enter" or "return" key) to the shell, and the OS responds by printing lines of characters back to the shell display.

When a user logs onto a system, a shell program is started to handle the interaction. Once the shell has initialized its data structures and is ready to start work, it clears the 25 line display, and then prints a prompt in the first few character positions on the first line. Sometimes the shell is configured to include the machine name as part of the prompt. The Linux development machine used here is named *kiowa* using a bash shell, so the shell prints

```
kiowa$
```

as its prompt string. The shell then waits for the user to type a command line in response to the prompt. The command line could be a string such as

```
kiowa$ ls  -al
```

terminated by an enter or return character (in UNIX, the character is represented internally by the NEWLINE character, '\n'). When the user enters a command line, the shell program makes the appropriate system calls to execute the command that appears on the command line.

Every shell has its own language syntax and semantics. In conventional UNIX shells a command line has the form *command   argument_1   argument_2   ...*

where the command to be executed is the first word in the command line and the remaining words are arguments expected by that command. As discussed in the attached document, the number of arguments depends on which command is being executed. For example, the directory listing command can be used with no arguments - by typing "*ls*," or it may have arguments prefaced by the "-" character, as in "*ls -al*" where "*a*" and "*l*" are arguments. Each command uses its own syntax for interpreting an argument. For example, a C compiler command might look like

kiowa$ cc -g -o deviation -S   main.c   inout.c -lmath

where the arguments "*g*", "*o*", "*deviation*," "*S*," "*main.c*," "*inout.c*," and "*lmath*" are all being passed as parameters to the C compiler, "*cc*." That is, the specific command determines which of the arguments may be grouped (like the "*a*" and "*l*" in the *ls* command), which arguments must be preceded by a "-" symbol, whether the position of the argument is important, and so on.

The shell relies on an important convention to accomplish its task: The command is usually the name of a file that contains an executable program (external command). For example, *ls* and *cc* are the names of files (stored in */bin* on most UNIX-style machines). In a few cases, the command is not a file name, but is actually a command that is implemented within the shell (internal command); for example *cd* ("change directory") is usually implemented within the shell rather than in a file. Since the vast majority of the commands are implemented in files, think of the command as actually being a file name in some directory on the machine. This means that the shell's job is to find the file, prepare the list of parameters for the command, and then cause the command to be executed using the parameters.

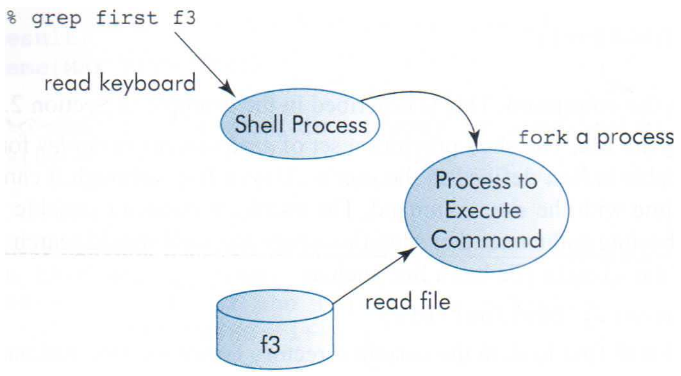
There is a long line of shell programs used with UNIX variants, including the original Bourne shell (sh), the C shell (csh) with its additional features over sh, the Korn shell, and so on, to the standard Linux shell (bash - meaning Bourne-Again shell). All of these shells follow a similar set of rules for command line syntax, although each has its own special features. The *cmd.exe* shell for Windows uses its own similar, but distinct, command language.

BASIC UNIX-STYLE SHELL DESIGN

A shell could use many different strategies to execute the user's computation. The basic approach used in modern shells is the one described in the attached document - to create a new process (or thread) to execute any new computation. For example, if a user decides to compile a program, the process interacting with the user creates a new child process. The first process then directs the new child process to execute the compiler program.

This idea of creating a new process to execute a computation may seem like overkill, but it has a very important characteristic. When the original process decides to execute a new computation, it protects itself from any fatal errors that might arise during that execution.

If it did not use a child process to execute the command, a chain of fatal errors could cause the initial process to fail, thus crashing the entire machine.
The UNIX paradigm for executing commands is illustrated in the following figure.



Here, the shell has prompted the user with the *%* character and the user has typed "*grep first f3*." This command means the shell should create a child process and cause it to execute the *grep* string search program with parameters *first* and *f3*. (The semantics of *grep* are that the first string is to be interpreted as a search pattern and the second string is a filename.)
The Bourne shell is described in Ritchie and Thompson's original UNIX paper. The Bourne shell and others accept a command line from the user, parse the command line, and then invoke the OS to run the specified command with the specified arguments. When a user passes a command line to the shell, it is interpreted as a request to execute a program in the specified file-even if the file contains a program that the user wrote. That is, a programmer can write an ordinary C program, compile it, and then have the shell execute it just like it was a normal UNIX command. For example, you could write a C program in a file named *main.c*, then compile and execute it with shell commands like

```
kiowa$ cc  main.c
kiowa$ a.out
```

The shell finds the *cc* command (the C compiler) in the */bin* directory, and then passes it the string "*main.c*" when it creates a child process to execute the *cc* program. The C compiler, by default, translates the C program that is stored in *main.c*, then writes the resulting executable program into a file named *a.out* in the current directory. In the second command, the command line is just the name of the file to be executed, *a.out* (without any parameters). The shell finds the *a.out* file in the current directory, and then loads it and executes it.

Consider the detailed steps that a shell must take to accomplish its job:

Printing a prompt: There is a default prompt string, sometimes hardcoded into the shell, e.g., the single character string "%," "#," ">" or other. When the shell is started, it can look up the name of the machine on which it is running, and prepend this string name to the standard prompt character, for example, giving a prompt string such as "*kiowa$*." The shell can also be designed to print the current directory as part of the prompt, meaning that each time the user employs *cd* to change to a different directory, the prompt string is redefined.

Once the prompt string is determined, the shell prints it to *stdout* whenever it is ready to accept a command line.

For example, this function prints a prompt:

```
void  printPrompt()
{
        /* Build the prompt string to have the machine name,  current directory, or other
           desired information */

        char promptString[] = ... ;
        printf (”%s”, promptString) ;
}
```

Getting the command line: To get a command line, the shell performs a blocking read operation so that the process that executes the shell will be blocked until the user types a command line in response to the prompt. When the command has been provided by the user (and terminated with a *NEWLINE* character), the command line string is returned to the shell.

```
void readCommand(char *buffer)
{
        /* This code uses any set of I/O functions, such as those in the stdio library to read
        the entire command line into the buffer. This implementation is greatly simplified,
        but it does the job. */

        gets (buffer);
}
```

Parsing the command: This is described in the example in the attached document.

Finding the file: The shell provides a set of environment variables for each user. This variable is first defined in the user's *.login* file, although it can be modified at any time with the *set* command. The *PATH* environment variable is an ordered list of absolute pathnames that specifies where the shell should search for command files. If the *.login* file has a line such as

```
set path=(.:/bin:/usr/bin)
```

the shell will first look in the current directory (since the first pathname is "." for the current directory), then in */bin*, and finally in */usr/bin*. If there is no file with the same name as the command (from the command line) in any of the specified directories, the shell responds to the user that it is unable to find the command. The solution needs to parse the PATH variable before it begins reading command lines.

This is done with:

```
int parsePath(char *dirs[])
{
        /*  This function reads the PATH variable for this environment, then builds an
            array, dirs[], of the directories in PATH  */

        char *pathEnvVar;
        char *thePath;

        for(i=0;  i<MAX_ARGS; i++)
                dirs[i] = …; /* set to null */
        pathEnvVar = (char *) getenv ("PATH");
        thePath = (char *) malloc(strlen(pathEnvVar) + 1);
        strcpy(thePath, pathEnvVar);

        /* Loop to parse thePath.  Look for a ':'  delimiter between each path name.  */
        …
}
```

The user may have provided a full pathname as the command name word, or only have provided a relative pathname that is to be bound according to the value of the *PATH* environment variable. If the name begins with a "/", then it is an absolute pathname that can be used to launch the execution. Otherwise, you will have to search each directory in the list specified by the PATH environment variable to find the relative pathname. Each time you read a command, you will need to see if there is an executable file in one of the directories specified by the *PATH* variable. The *lookup()* function is intended to serve that purpose:

```
char *lookupPath(char  **argv, char  **dir)
 {
        /* This function searches the directories identified by the dir argument to see
           if argv[0] (the file name) appears there.
          Allocate a new string, place the full path name in it, then return the string. */

        char *result;
        char pName[MAX_PATH_LEN];


        /* Check to see if file name is already an absolute path name */
         if(*argv[0] == '/') {
        …
        }

        /*  Look in PATH directories.
            Use access() to see if the file is in a dir.*/
        for(i = 0; i < MAX_PATHS; i++) {
                ...
        }
        /* if file name not found in any path variable till now then*/
        fprintf(stderr, "%s: command not found\n", argv[0]);
        return NULL;
}
```

ATTACKING THE PROBLEM

Begin your solution by reading the attached introductory document. You will have to rewrite that code to support user interaction. Here is **a header file**, *minishell.h*, for the mini-shell:

```
#define         LINE_LEN        80
#define         MAX_ARGS        64
#define         MAX_ARG_LEN     16
#define         MAX_PATHS        64
#define         MAX_PATH_LEN    96
#define         WHITESPACE       " .,\t\n"

#ifndef NULL
#define NULL  ...
#endif

struct command_t {
        char *name;
        int argc;
        char *argv[MAX_ARGS];
};
```

Here is **the skeleton** of a solution:

```
/*  This is a very minimal shell. It finds an executable in the PATH, then loads it and
executes it (using execv). Since it uses "." (dot) as a separator, it cannot handle file
names like "minishell.h". */

#include  …

#include "minishell.h"

char *lookupPath(char **, char **);
int parseCommand(char *, struct command_t *);
int parsePath(char **);
void printPrompt();
void readCommand(char *);
…
```

```
int main()
{
        …

        /* Shell initialization */
        …

        parsePath(pathv); /* Get directory paths from PATH */

        while (TRUE) {
                 printPrompt();

                /* Read the command line and parse it */
                readCommand(commandLine);
                …
                parseCommand(commandLine, &command);
                …

                /* Get the full pathname for the file */
                command.name = lookupPath(command.argv, pathv);
                if(command.name == NULL) {
                        /* Report error */
                        continue;
                }

                /* Create child and execute the command */
                 …

                /* Wait for the child to terminate */
                …
        }

        /* Shell termination */
        …

}
```