Project #1:
Command-Line User Interface Shell and Utilities

Principles of Operating Systems (LAB)

COP 4610/CGS 5764 (Fall 2014)

Due Date and Time:

Refer to the lab calendar at: http://www.cs.fsu.edu/~cop4610t/calendar.html.

1 Overview

For this project you will implement a *command-line (text-based) user interface shell*. The purpose of this shell is to provide a means for users to interact with an operating system (OS).

Your shell is to provide a subset of the UNIX utilities as well as some non-UNIX ones. The behavior of the UNIX utilities (denoted in this document by the `UNIX` tag next to their name) should adhere to the *Single UNIX Specification* (SUS) standards published by *The Open Group*. The additional non-UNIX utilities and features (identified by the `NON-UNIX` tag next to their name) are detailed in this document and during the lab lecture(s).

2 Objectives

The objective of this project is to develop an initial understanding of the components of an OS and approaches to implement them. Upon successful completion of this project, the student should possess the following understanding and skills.

- Understand one commonly used process model to execute program instructions.
- Understand the effect of practical factors on implementing OS utilities.
- Commonly used means to communicate with both the OS and the user.
- Reasonable ability to implement OS functionality using C.
- Low-level programming development and debugging skills.

3 Required Operation

The following details the required behavior of the implemented shell.

3.1 Prompt NON-UNIX

The shell should output a prompt when it can accept new commands. The format of the prompt text should appear as follows:

`<username>@<hostname>:<working_directory>␣$␣`

where the angle brackets indicate the text that should substituted depending on the state of the system and '␣' signifies a literal space (whitespace).

> `working_directory` is the *absolute* path (i.e., starting from the root) of the shell's current working directory. For example, if the binary `pwd` is executed by the shell, it should be the same as the `working_directory` displayed in the prompt.
>
> `hostname` is the name of the machine executing your shell. The host name can be acquired using the `gethostname()` function.
>
> `username` is the name of the logged in user. The username can be acquired using the `getlogin()` function.

Example

`cop4610t@linprog:/home/grads/cop4610t $`

3.2 Program Execution UNIX

When the user enters the name of a binary program that is specified by an absolute path or is located in its search path, the shell should execute it. The search path is identified by the `PATH` environmental variable, which contains a list of directories separated by a colon (e.g., `/usr/local/bin:/usr/bin`. When determining if the user-entered program file is located in a directory specified by the `PATH` environmental variable, the search should start with the directory whose first character is at position zero of the `PATH` string. The search should stop when the first match is found.

As an example, typing "`ls`" or "`grep`" and pressing `Enter` at the prompt should execute the binary program as a separate process.

Your shell should not terminate after executing any command or utility, but wait until the executing binary completes, unless stated otherwise (e.g., back-

ground execution). Once the program completes, a prompt should be displayed on the screen.

As a reference you can also view the SUS standard:

http://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html#tag_18_09_01_01

Environmental variables (SUS standard reference):

http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap08.html#tag_08

`getenv()`

http://pubs.opengroup.org/onlinepubs/9699919799/functions/getenv.html#

Background Execution UNIX

If an ampersand (&) is at the end of command line entered by the user (i.e., the last non-whitespace character before enter is pressed), the command should be executed in the background. That is, your shell should instruct the OS to start the program (with the appropriate configuration) and immediately present the user with a prompt to enter further commands. The user should be able to type in and execute new commands while the previous command is executing (if it has not yet exited).

3.3 Input/Output Redirection UNIX

http://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html#tag_18_07

Input UNIX

```
cat < a.txt
```

Output UNIX

```
ls > ls.txt
```

Appending Redirected Output UNIX

```
ls >> ls.txt
```

Pipelines UNIX

```
ls | grep ‘‘a’’ | grep ‘‘b’’
```

3.4 Built-In Utilities

`exit [n]` UNIX

Shall cause the shell to exit and return the integer value `n` to the OS as the exit status. If `n` is not provided, the exit status is assumed to be zero.

http://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html#exit

`cd [directory]` UNIX

Changes the current working directory of the shell to the path specified by the `directory` operand. `cd` supports relative (directories relative to the current working directory prior to executing `cd` using '.', and '..') and absolute directories, as well as the special home directory '~'.

If a <hyphen> is specified as the operand, an action equivalent to the following should occur:

> `cd "$OLDPWD" && pwd`
>
> where `$OLDPWD` is the working directory prior to the execution of the most recent `cd` command.
>
> In other words, change to the previous working directory and then output the its path.

If no `directory` operand is specified, the `cd` utility shall behave as if the user's home directory is specified as the `directory` operand.

`ioacct <command line>` NON-UNIX

Executes the rest of the `command line` operand as if it were entered directly on the shell command line. Once execution of the `command line` completes, it will print the number I/O bytes read and written. `ioacct` does not have to work with background processes, pipes, or input/output redirection.

The I/O information for a particular `pid` can be obtained by viewing the `read_bytes` and `write_bytes` entries in the proc file `/proc/<pid>/io`. See
http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/filesystems/proc.txt?id=v3.16#n1443
for documentation on the `io` file's contents.

Example Output of /proc/<pid>/io

```
rchar:  1900
wchar:  0
syscr:  7
syscw:  0
read_bytes:  0
write_bytes:  0
cancelled_write_bytes:  0
```

3.5 User Input Error Handling

The following user input errors should be detected by the shell and an appropriate error message displayed to the user. Following the error message, the user should be presented with a prompt.

- Malformed Input/Output Redirection
- Incorrectly Placed Ampersand
- Non-Existent Paths (e.g., file locations when executing a command, directories passed to `cd`)

3.6 Assumptions

The following are assumptions you can make about the user-entered commands (i.e., the following will not be used as test cases for grading):

- No more than 2 pipes (i.e., "|") will be part of a single command.
- Both input and output redirection (i.e., "<" and ">" will not appear together in a single command.
- You do not need to handle Wildcard expansion (e.g., `ls *.c`), escaped strings (`cat hello\ world.txt`), and quoted strings (`cat ‘‘hello world’’`).
- No more than 255 characters per complete command (before hitting return) will be entered by the user.

4 Restrictions

- The code must be written in C and compile with the version of `gcc` installed in the lab.
- `system()` library call may not be used
- `execv()` is the only call of the `exec*()` family that may be used (e.g., use of `execvp()` is not permitted)

5 Extra Credit

Support multiple pipes with a loop that allows your shell to support an arbitrary number of pipes limited only by the OS and hardware.

A novel, reasonably useful utility of your choosing. You must provide a written specification on the proper operation of your novel utility.

6 Submission

The following is a list of files expected for a project submission:

- Source code and `Readme` files
- `Makefile` that builds the shell binary when executing the `make` command in the same directory as the source code files.
- Project report as explained in the lab lecture and outlined in the associated slides.

One person from the team should submit a gzip'd tarball file containing the above files to Blackboard. The `source code` and `Makefile` should be placed inside a "`src`" directory (**Do not** submit binaries).

The filename of the submission file is to be formatted as: "`<group name>.Project1.tar.gz`" `tar` and `gzip` of the files which can be accomplished using the following command:

```
tar cfzv CookieMonster.Project1.tar.gz *
```

7 Example Output

```
cop4610t@linprog:/home/grads/cop4610t $ ioacct cat my_file.txt > /dev/null
bytes read: 45
bytes written: 0
cop4610t@linprog:/home/grads/cop4610t $ ls
hello.txt  text.txt
cop4610t@linprog:/home/grads/cop4610t $ ls | grep test
test.txt
cop4610t@linprog:/home/grads/cop4610t $ exit 55
```

Your code should return the exit status of 55, which can be checked by running the command `echo $?` from the shell that you used to start your program.

```
echo $?
55
```

8 Grading

Item	Points (out of 100)
Report	25
Code quality and Readme	10
Prompt • correctly displayed text • displayed at correct events (e.g., after completion of command)	7.5
Execute programs • Properly handles path. • Can your shell start another instance of itself?	10
Input Redirection	5
Output Redirection	5
Pipelining • Handles 2 pipes in a single command line.	10
`cd <operand>` operand type • relative directory • absolute directory • hyphen • none	10
`exit`	2.5
`ioacct`	10
User Input Error Handling	10

Bonus

Support "unlimited" pipes Novel utility	5

Deductions

Memory Leak(s)	-10
Zombie (defunct) process(es)	-5
Fails to compile when using `make` and your `Makefile`	-25
Late submission per each 24-hour period after due time	-5 per 24-hours