CSC 246 Fall 2019 Homework 1

Due: January 28 2019, 11:55PM

ChangeLog

01/17: A link to "SSH Secure Shell Client" is provided. "The maximum number of command arguments for execlp() should be 6" in Problem 3 is changed to "The maximum number of arguments for execlp() should be 6" to be consistent with the following sentence. Problem 3 now provides a sample Makefile.

01/15: "remote.eos.ncsu.edu" can also be "remote-linux.eos.ncsu.edu". All three subproblem under problem 1 are further clarified. More instructions on how to access manual pages are added. A wiki link for file descriptors is added.

01/15: first version online

Instructions

(a) You can work on any current Linux installation. We will use the EOS Linux machines (one of the Linux desktop machines in EB 3 or Daniels, or via remote.eos.ncsu.edu or remote-linux.eos.ncsu.edu) to do the grading. If you are using Linux, type "ssh [unity_id]@remote.eos.ncsu.edu" or "ssh [unity_id]@remote-linux.eos.ncsu.edu" to access the machines remotely. For Windows, use "putty" to ssh into remote.eos.ncsu.edu or remote-linux.eos.ncsu.edu. You can also use SSH Secure Shell Client instead of putty, which further includes a GUI for sftp. Use your campus password to get through.

(b) Makefile and README: Use Makefile to compile all source files of your code. Use gcc as your compiler. Also write a README explaining what your code does and how to compile and execute it. The README is in addition to the comments in your source files at critical points in your code explaining what does that piece of code do. You can refer to the lab tutorial chapter of the OSTEP book for some help on Makefile.

(c) File names: For programming problems, name your source files as prob_x.c and header files as prob_x.h, where x is the question number here. Try to just use one .c and one .h file for each problem. For non-programming problems, put the anwser to one single ASCII format text file and name it problems.txt. Please submit ASCII format text file ONLY for non-programming problems. You may be penalized for incorrect filenames or formats.

(d) The first line of each file: You MUST put your name and unity id on the top of each file you submit, formatted as follows:
/* unity-id FirstName MiddleInitial LastName */
Don't leave any field blank.

(e) Before you submit: *** IMPORTANT*** Please make sure your programs compile and execute normally on the specified VCL image, you will receive 0 for programs that do not compile or do not execute. You will receive partial credit for programs that do not produce correct output.

Problem 1

In this problem, we are going to use the tool, strace, to track system calls invoked while another system tool, cat, is executed, where "cat" is a program that can be used to print the content of files to the screen. A sample file is provided.

Please use the following command to redirect and save the output:
"strace -o strace_result cat sample_file"
Read the strace_result file and answer the following questions:

1. Among all the system calls invoked, which ones read data from the file (i.e., sample_file) and which ones write data to the output screen device? Show the system calls from the strace_result file, and please include the parameters.
2. For the system calls identified in subproblem (1), what are the corresponding files of the file descriptors appearing in these system calls? Note that stdout and stderr are legit possible answers. If you need help on understanding what is a file descriptor, check this wiki page. Make your best guess.
3. How would you use the strace tool to determine the values of file descriptors for stdout and stderr?

Put your answers in problems.txt (ASCII file).

Problem 2

For this question, read the manual pages for fork() and waitpid() carefully. To read the manual page for a particular command, function, or system call, you can usually just type "man command/fuction/system_call". If two different manual pages have the same name, you can specify the section of the manual to make sure you get the page you want. For example, "man 1 stat" will give you documentation for the stat shell command, and "man 2 stat" will tell you about the stat() system call. For fork and waitpid, "man fork" and "man waitpid" are suffice. You can certainly search the internet, as manual pages are also stored by many different sites. It is just that typing commands in a terminal can get you comfortable with the terminal quicker, as you will be using it a lot throughout the semester.

Consider a child process ProcC created by process ProcP using fork(). ProcP waits for ProcC using waitpid() after ProcC is created, assuming ProcC exits after ProcP calls waitpid(). Initially, ProcP has opened one file, f1, with "rw" access using open(). After ProcC exits, ProcP closes the file and exits. Answer the questions based on the information in the manual pages, and the manual pages should have all the information you need.

1. Can ProcC access the file opened by ProcP? Please reference necessary sentences from the manual pages that can support your answer.
2. Suppose ProcP writes some data D1 on f1 before creating ProcC and some data D2 after creating ProcC. Which data will be visible to ProcC and why? Can ProcC write anything on f1? Please reference necessary sentences from the manual pages that can support your answer.
3. What will happen to ProcC if ProcP does not wait on ProcC? Can ProcP wait on ProcC after ProcC has exited? What happens if ProcP exits as well without waiting on ProcC at all? Please reference necessary sentences from the manual pages that can support your answer.
4. How can ProcP check the return value of ProcC? Please reference necessary sentences from the manual pages that can support your answer.

Put your answers in problems.txt (ASCII file).

Problem 3

In this problem, you are asked to implement a mini shell program, through which external programs can be executed from command lines. As shells you are using on Linux, our mini shell first parses each user command, and it then executes each command in a separate process. However, being a mini shell, it will be less sophisticated, and there is no need to support features like input/output redirection, pipes, or background execution. Completing this project will involve using the fork(), execlp(), and waitpid() system calls. Note that we are specifically requiring the use of execlp() among the exec system function family.

The Core

A shell interface gives the user a prompt ("> " in our case), after which the next command is entered. One technique for implementing a shell interface is to have the parent process first read what the user enters on the command line and then create a separate child process that performs the command. This will require parsing what the user has entered into separate tokens and storing the tokens in an array of character strings. For example, if a user enters the command "ls -al", the values stored in the array are: "ls", "-al", and NULL. This args array will be used by the execlp() function, and please check the manual page to learn how to use the function.

You are required to have the parent process wait and track the child process's return status using waitpid() system call, see the WIFEXITED() and WEXITSTATUS() macros (read the man pages for their detailed usage). For other error conditions that you catch and handle, print out error messages using your own format. The mini shell exits on the "exit" command.

Aliases

You are also asked to allow users to add their own command names as aliases. Here is an example: normally, one types "ls -al" to show the extended information of all files in the current directory. If you type "alias ll ls -al" in the mini shell, it will establish an alias "ll" for "ls -al". The next time when you enter "ll", you will get the same results as typing "ls -al".

The maximum alias table entry number is required to be 8, which excludes duplicated entries. If the command is "alias" without extra arguments, please show all the aliases stored in the alias table. An "alias" with one argument just prints the matching alias stored in the table, or an error message if it does not exist.

The maximum number of arguments for execlp() should be 6, which includes the program name and the last argument NULL (should be used as the last argument). Your alias should take care of partial replacement cases of arguments, for example:

```
> alias <a> <b> <c>
> <a> <d>
```

The command now is actually <b> <c> <d> instead of only <b> <c>.

You can first parse your command and check if it is an alias command starting with the string "alias". If it is an alias command with no argument, print all aliases; if it is an alias command with one argument, print the matching alias; and if it has two or more arguments, create a new alias if the alias is not taken yet. If it is not an alias

command, you first try to determine if it is a match of any command with aliases defined. Upon a match, the mini shell expands the command accordingly. Finally, the mini shell uses fork() and execlp() to execute the command.

sample test cases:

```
./prob_3
> alias ll ls -al
> ll
drwx------ 2 user ncsu  2048 Aug 26 11:47 .
drwx------ 5 user ncsu  2048 Aug 26 01:43 ..
-rw------- 1 user ncsu   353 Aug 26 09:16 Makefile
-rwx------ 1 user ncsu 14027 Aug 26 09:41 prob_3
-rw------- 1 user ncsu  5343 Aug 26 09:42 prob_3.c
> alias c cat
> c Makefile
# the compiler: gcc for C program, define as g++ for C++
CC = gcc
RM = rm
# compiler flags:
#  -lm    for utilizing math.h
#  -I     Include path
CFLAGS = -lm
IFLAGS = -I
# the build target executable:
EXECUTABLE = prob_3
all: $(EXECUTABLE)
$(EXECUTABLE): $(EXECUTABLE).c
        $(CC) -o $(EXECUTABLE) $(CFLAGS) $(EXECUTABLE).c
clean:
        $(RM) $(EXECUTABLE)
> alias
ll      ls      -al     (null)
c       cat     (null)
> alias ll
ll      ls      -al     (null)
> alias mm
ERROR: no such alias
> c <non_existing_file>
cat: <non_existing_file>: No such file or directory
ERROR: cat return 1
> cat <non_existing_file>
cat: <non_existing_file>: No such file or directory
ERROR: cat return 1
> exit
```

Miscellaneous

If the input only includes spaces, tabs, or Enters, your terminal should not terminate or crash.

You are given a programming skeleton prob_3.c with parsing code, which you do not need to follow. If you decide to follow, there could lines that need to be modified based on the specification above. The skeleton also provides the general operations of a command-line shell. The main() function presents the prompt "> " and outlines the steps to be taken after input from the user has been read. The main() function continually loops until the user enters exit at the prompt.

A sample Makefile is provided, under which you can do "make" (equivalent to "make all"), "make prob_3", and "make clean". After you finish, turn in your source files, Makefile, and README.