CS342 Operating Systems - Spring 2021
Project #1 – An intercepting shell program (ISP)

Assigned: Feb 14, 2021, Sunday.
Due date: Feb 28, 2021, Sunday, 23.59. Document version: 1.0

This project will be done individually. You will use the C programming language in Linux OS.

Shell Program (80 pts)

In this project you will develop a simple shell program, a command line interpreter, called isp (intercepting shell program). Your shell will provide a prompt string to a user (like isp$), where the user will type a command and your shell will execute the command.

A command to execute will include a command name, i.e., a program name, and zero or more parameters. An example command can be "cp file1.txt file2.txt". When such a command is entered by a user, your shell will divide it into arguments, i.e., strings, where argument 0 is the command name, argument 1, if any, is the first parameter to the command, argument 2 is the second parameter to the command, and so on.

When started, your shell will run as a process, i.e., main (parent) process, and will wait for an input command line. When user gives an input line, the line will be separated into arguments and a child process will be created to execute the command. For this, the main process will use the **fork**() system call. In the child process, **exec** system call will be used to finally execute the program. There are various exec related library functions. You can use execv or execvp, for example. The exec system call takes the pathname of a program to execute and an array of argument strings. Please read the manual (man) pages of the related functions. After creating the child process, the parent process will wait. When command is executed and child process terminates, the parent process will return from waiting and will provide another prompt string to the user so that the user can type another command line.

Your shell will also support composition of two commands where the output of one command will be given as input to another command. For example, there is "ps -aux" program in Linux that is listing the current processes in the system, and there is "sort" program in Linux that is sorting a text file. When we write "ps aux | sort" in Linux shell, it prints the sorted list of processes to screen. Similarly, when we would write such a command line in your shell, it should also print a listing of processes in sorted order. Such a command line consists of two commands, with possible parameters, separated with | symbol. The symbol | is called the pipe symbol. Your shell will support use of only *one* pipe symbol in a command line, hence the compound execution of *two* commands in a command line.

When a command line with pipe symbol is entered in your shell, the main shell process with create two child processes. Two fork calls are needed to do this. In each child we need to use the exec system call to execute the respective program. The output of the first program, that would normally go to screen, i.e., to standard output, has to go now as input to the second program, which would normally receive the input from a user. This requires communication (IPC) between these processes. Your shell will provide communication in one of two modes.

In first mode, **normal mode**, a single unnamed Linux pipe will used for communication, and it will be used directly by the child processes. That means, the output of one child is fed directly to the pipe from where the second child will get the data. You will enable this by *I/O re-direction*.

In the second mode (**tapped mode**), two Linux unnamed pipes will be used and data will flow indirectly between child processes. The main process will be on the data flow path. The main process will create two pipes. The output of the first child process will be directed to the first pipe, from where the main process will read the incoming stream of bytes (characters). The main process will write those characters to the second pipe from where the second child will take the input. The main process will read from first pipe and write to second pipe **N** characters at a time using read and write system calls. N is a value that is given as an argument to the shell program when it is started. N can be between 1 and 4096. Hence your shell will have the following parameters:

isp <N> <mode>

where <N> is the number of bytes to read/write in one system call and <mode> is the mode of the communication to use. If mode value is 1, then normal communication mode is used. If mode value is 2, tapped communication mode is used.

An example invocation can be: "./isp 1 2". That means, we want to read/write 1 byte at a time and tapped mode is used.

The main process will create the pipes using the **pipe**() system call. After creating the pipes, the main process will create the child processes using the fork system call. Then exec system call will be called in each child to run the respective program. Before calling exec, in each child, *I/O redirection* will be done to re-direct the output or input. For the first child, the output will not go to standard output anymore but will be directed to first pipe. For the second child, the input will not come from standard input anymore but will be taken from the second pipe.

I/O re-direction can be done by use of the **dup2()** function. This function is used to duplicate a file (I/O) descriptor (i.e., copy one descriptor to the other so that the other behaves as the first one). For the first child, the write-end of the first pipe will be duplicated to file descriptor with number (index) 1. That means, the write-end descriptor of first pipe will be copied to standard output descriptor in the open file table of the process. In the open file table for a process, the *standard output* descriptor index is always 1. The dup2 call will make the write-end of first pipe to act as standard output. In this way whenever child 1 program writes data to standard output, the data will go to the first pipe through the write-end descriptor. This redirection can be enabled by the statement: dup2 (pipe1[1], 1), assuming pipe1 is an array storing the descriptors of the first pipe.

For the second child, the read-end of second pipe will be duplicated to file descriptor 0. Integer 0 is always the file descriptor (index) corresponding to *standard input*. This can be done by the statement: dup2 (pipe2[0], 0), assuming pipe2 is an array storing the descriptors of the second pipe. When we do this, the read-end of the second pipe will act as the standard input for child program 2. Then, whenever program 2 reads from standard input, it will now read from the second pipe, instead of keyboard.

After creating child processes, the main process will read from the first pipe and will write to the second pipe. Do not forget to close the unused ends of the pipes at the main process. When a child terminates, the ends of the pipes that are used by the child are closed automatically as well.

For compound command execution, since the main process is on the data flow path from one child process to the other, it can keep some statics about the transferred data. We can count the number of bytes transferred through the pipes for compound commands. We can also count the number of read/write operations performed from and to pipes. Please print out these counts to screen after compound command execution has finished. The output format should be as in the following example:

character-count: 15000
read-call-count: 15000
write-call-count: 15000

I/O redirection in the first mode, i.e., normal mode, will be done similarly, again by using dup2 function. Note that in this case the main process will not intercept, i.e., will not be on the data flow path, and therefore the second child will read from the same pipe that the first child is writing into. Therefore, please duplicate the descriptors (i.e., use dup2) considering this in mind.

Experiments (20 points)

First write two simple programs "producer M" and "consumer M" as two programs to be compounded. When separately executed, the producer will just print M random alphanumeric characters to screen one-by-one (one character at a time), and consumer will just read M characters from standard input (keyboard) one-by-one.

Do timing experiments to compare normal mode execution time with tapped mode execution time. That means, for a compound command, find out how long it takes to finish a compound command (in

this case *producer M | consumer M*) in normal mode and in tapped mode. Also analyze the effect of the N parameter of your shell. Do experiments for this purpose as well. Use various M values Plot your results. Interpret the results and try to draw conclusions. You can use the gettimeofday function to measure the time in your program.

Put all these into your report.

Submission

Submit a pdf file as your report presenting and discussing your experiments. Your report will include the results, your interpretations and conclusions. Put your report.pdf file and all other files (isp.c, Makefile, README.txt file, producer.c, consumer.c) into a directory named with your ID. Then tar and gzip the directory. For example a student with ID 21404312 will create a directory named "21404312" and will put the files there. Then he/she will tar the directory (package the directory) as follows:
tar cvf 21404312.tar 21404312

Then he/she will gzip the tar file as follows:
gzip 21404312.tar

In this way he/she will obtain a file called 21404312.tar.gz. Then he/she will upload this file into Moodle.

Tips and Clarification

- Starting early is highly recommended.
- Work incrementally.
- Learn how to use a debugger (gdb, xxgdb, or something else).
- Be careful about strings and memory allocation (malloc) for them. Do *not forget to close* a pipe descriptor (read end or write end) that is not used in a process (parent or child). This is important. Note that after creating a pipe, if you do fork(), all descriptors are copied to child as well and they are available for child's use as well. After all processes close the write end of a pipe, an EOF can be received from the other end (receiving end). In this way receiving end can understand the end of a character stream. This is important.
- A skeleton code is put into github. You can start with it if you wish.
  https://github.com/korpeoglu/cs342spring2021-p1