CS111 - Project 1A: I/O and IPC

INTRODUCTION:

In this project, you will build a multi-process telnet-like client and server. Part A of the project (this part!) can be broken up into two major steps:

- Character-at-a-time, full duplex terminal I/O
- Passing input and output between two processes

You will be extending the same code in Project 1B, so make sure it's readable and easy to modify.

RELATION TO READING AND LECTURES:

This lab will build on the process and exception discussions in lecture 3, but it is really about researching and exploiting APIs.

PROJECT OBJECTIVES:

- Demonstrate the ability to research new APIs and debug code that exploits them
- Exploit the following OS features:
    - Terminal I/O and modes (as an example of a complex API set)
    - Inter-process communication
    - Exception handling
- Develop a multi-process application

DELIVERABLES:

A single tarball (.tar.gz) containing:

- a single C source module that compiles cleanly (with no errors or warnings).
- a Makefile to build the program and the tarball.
- a README file describing each of the included files and any other information about your submission that you would like to bring to our attention (e.g. limitations, features, testing methodology, use of slip days).

PROJECT DESCRIPTION:

You will write a program that compiles to an executable named *lab1a*, and will accept the command line argument *--shell* (explained below).

0. Study the following manual sections:
    - ○ *ioctl(2)* ...to manipulate the settings of I/O devices generally
    - ○ *termios(3)*, *tcgetattr(3)*, *tcsetattr(3)*, etc. ...for manipulating terminal attributes
    - ○ *fork(2)* ...for creating new processes
    - ○ *waitpid(2)* ...to allow one process to monitor another process's state, and react to changes in state
    - ○ *pthreads(7)* ...for multi-threaded applications
    - ○ *exec(3)* ...a family of calls for executing files within a process
    - ○ *pipe(2)* ...for inter-process communication
    - ○ kill(3) ...for sending signals to processes by PID
    - ○ *atexit(3)* ...not required, but may be useful for reverting the console back to its original state (in part 1 below)
1. Character-at-a-time, full duplex terminal I/O: write a program to
    - ○ put the console into character-at-a-time, no-echo mode (also known as non-canonical input mode with no echo).
    - ○ read input from the keyboard into a buffer. Do something reasonable if you exceed the buffer size.
    - ○ map received <cr> or <lf> into <cr><lf> (see below for a note on carriage return, linefeed, and EOF).
    - ○ write the received characters back out to the display, one character at a time.
    - ○ upon detecting a pre-defined escape sequence (^D), restore normal terminal modes and exit (hint: do this by saving the "normal" terminal settings and reverting to these on exit).
2. Passing input and output between two processes: extend part 1 to support a --shell argument to pass input/output between the terminal and a shell:
    - ○ fork to create a new process, and then exec a shell (/bin/bash), whose stdin is a pipe from the terminal process, and whose stdout/stderr are (dups of a single) pipe to the terminal process. (You will need two pipes, one for each direction of communication, as pipes are unidirectional)
    - ○ one thread should read input from the keyboard, echo it to stdout, and forward it to the shell. <cr> or <lf> should echo as <cr><lf> and go to shell as <lf>.
        - i. another thread should read input from the shell pipe and write it to stdout.
        - ii. if you'd like, you can do this with only one extra thread, as the parent process is itself a thread.
    - ○ upon receiving an interrupt (^C) from the terminal, send a SIGINT to the shell
    - ○ upon receiving an EOF (^D) from the terminal, close the pipe, send a SIGHUP to the shell, restore terminal modes, and exit with return code 0.
    - ○ upon receiving EOF from the shell, restore terminal modes and exit with return code 1.
    - ○ upon receiving a SIGPIPE from the shell (i.e., the shell exits), restore terminal modes and exit with return code 1. *****UPDATE***: the return code has been changed so that SIGPIPE or EOF from shell result in the same behavior.**
    - ○ print the shell's exit status when the program closes (hint: use *waitpid(2)* and associated functions)

SUBMISSION:

Project 1A will be due on Wednesday, October 5.
Your tarball should have a name of the form lab1a-*studentID*.tar.gz and should be submitted via CCLE.
We will test it on a SEASnet GNU/Linux server running RHEL 7 (this is on lnxsrv09). You would be well advised to test your submission on that platform before submitting it.

A NOTE ON CR, LF, AND EOF:

Long ago, when interaction was through mechanical teletypes, these ASCII codes had the following meanings:

0x0D	carriage return, meant move the printing head back to the left edge of the paper.
0x0A	line feed, meant move the paper upwards one line
0x04	end of file, meant there is no more input.

So every line of text ended with <cr><lf> or 0x0D 0x0A. This is still the case in Windows.

Other people felt it was archaic foolishness to require two characters to indicate the end of line, and suggested that <lf> 0x0A (renamed newline) should be all that was required. This is how files are represented in UNIX descendants. But when output is sent to a virtual terminal, the newline is still translated into the two distinct motion characters <cr> and <lf>.

DOS systems used to actually put a 0x18 (^Z) as the last character of a file to indicate END OF FILE, while most other operating systems simply ended the data (subsequent reads return nothing).

RUBRIC:

Value	**Feature**
	Packaging and build (15%)
5%	untars expected contents
5%	clean build w/default action (no warnings)
3%	Makefile has clean and dist targets
2%	reasonableness of README contents
	Basic functionality (25%)
10%	Correctly changes console to char-at-a-time, no-echo mode
10%	Keyboard input is written to the console one character at a time
5%	Handles long inputs
	With the --shell option (45%)
10%	forks a process for the shell
10%	in one thread (may be the parent process), correctly pass keyboard input to stdout and to shell
10%	in a separate thread, read input from shell pipe and echoes to stdout
3%	On ^D: sends SIGHUP to shell, RC = 0
3%	On ^C: sends SIGINT to shell
3%	SIGPIPE from shell to parent, RC = 1
3%	EOF from shell: exit, RC = 1
3%	Report shell's exit status on exit
	Miscellaneous (15%)
5%	Correctly interprets <lf> and <cr>
10%	Restores terminal modes on program exit