Project 1A
Terminal I/O and Inter-Process Communication

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
- Polled I/O and passing input and output between two processes

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

RELATION TO READING AND LECTURES:

This lab will build on the process and exception discussions in the lecture on processes and exceptions, 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)
  - Polled I/O (a means of awaiting input from multiple sources)
  - Inter-process communication
  - Exception handling
- Develop a multi-process application
- Develop debugging skills for multi-process and non-deterministic problems

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 (compiling it with the `-Wall` and `-Wextra` options) and the tarball. You are not required to include a `check` target in this Makefile, but it should support the standard default, `clean` and `dist` targets.
- 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., research, limitations, features, testing methodology).

PROJECT DESCRIPTION:

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

1. Study the following manual sections:
   - *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
- *exec(3)* ... a family of calls for loading a new program into a running process
- *pipe(2)* ... for inter-process communication
- *kill(3)* ... for sending signals to processes by PID
- *strerror(3)* ... descriptions associated with system call errors
- *poll(2)* ... to wait for the first of multiple input sources

2. Character-at-a-time, full duplex terminal I/O

   Write a program (executable should be called `lab1a`) to:
   - put the keyboard (the file open on file descriptor 0) into character-at-a-time, no-echo mode (also known as non-canonical input mode with no echo). It is not suggested that you attempt to construct a set of correct terminal modes from scratch, as there are numerous subtle options, most of which will render your terminal window unusable if incorrectly set. Rather, it is suggested that you get the current terminal modes, save them for restoration, and then make a copy with only the following changes:

     ```
     c_iflag = ISTRIP;       /* only lower 7 bits    */
     c_oflag = 0;            /* no processing        */
     c_lflag = 0;            /* no processing        */
     ```

     and these changes should be made with the `TCSANOW` option. Note that this sort of mode change may result in the flushing of queued input or output characters. Thus the modes should be set immediately on start-up (before any characters are entered) and reset as the last thing before shutdown (after all returned output has been processed).
   - read (ASCII) input from the keyboard into a buffer. In character-at-a-time mode, you get characters as soon as they are typed, without waiting for a newline. In most cases you will only get one character for each read, but if the system is running slowly and you type fast multiple characters may have accumulated by the time you issue your *read*. Thus you should do a larger read (for whatever your buffer size is) and process all the characters you receive.
   - 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, as they are processed.
   - upon detecting a pre-defined escape sequence (^*D*), restore normal terminal modes and exit (hint: do this by saving the normal terminal settings when you start up, and restoring them on exit).

   Note: You will surely have your program exit without properly restoring modes, after which your terminal session may seem unusable. Get familiar with the `stty sane` option of *stty(1)*. If your program exists with your terminal modes in some unusable state, you can usually fix it by typing ^*J* `stty sane` ^*J*, where ^*J* is a *control-J* (or linefeed character).

3. Passing input and output between two processes

   Extend and refactor your program to support a `--shell=`*program* argument to pass input/output between the terminal and a shell:
   - fork to create a new process, and then exec the specified program (with no arguments other than its name), whose standard input is a pipe from the terminal process, and whose standard output and standard error are (dups of) a pipe to the terminal process. (You will need two pipes, one for each direction of communication, as pipes are unidirectional.)

     On most systems the most appropriate shell is probably `/bin/bash`.

   - read (ASCII) input from the keyboard, echo it to stdout, and forward it to the shell. <*cr*> or <*lf*> should echo as <*cr*><*lf*> but go to shell as <*lf*>.

- read input from the shell pipe and write it to stdout. If it receives an <*lf*> from the shell, it should print it to the screen as <*cr*><*lf*>.

  We type one character at a time, and so each *read(2)* from the keyboard is likely to return only one character, even if we do the *read* for a much larger number of characters. But a shell command may generate many lines, so reads from the shell are likely to return many more characters. It is suggested that you should do a large (e.g., 256 byte) *read*, and then process however many characters you actually receive.

- Data should be passed in both directions as soon as it is received. Do not wait until you encounter a newline. The shell can process character-at-a-time input, and many commands generate output that is not followed by a newline.

The trick here is that will not be a strict alternation between input from the keyboard and input from the shell. Either is capable of generating input at any time. If we do a *read(2)* from the keyboard, we will block until keyboard input becomes available, even if there is input from the shell pipe waiting for us! The *poll(2)* system call enable us to solve this problem:

- Create an array of two `pollfd` structures, one describing the keyboard (stdin) and one describing the pipe that returns output from the shell.
- Have both of these `pollfd`s wait for either input (`POLLIN`) or error (`POLLHUP`, `POLLERR`) events.
- Write a main loop that calls *poll(2)* (`timeout=0`) and then only reads from a file descriptor if it has pending input (as reported in the corresponding `revents` field).
- It is possible that by the time we do our final *poll* the shell will have both returned its final output and exited, so that there will be multiple events (e.g. `POLLIN`, and `POLLHUP`) associated with the pipe from the shell. Always process all available input before processing the shut-down indication. Otherwise you may discard the final output.

You should now be able to type shell commands to your program, see them echoed as you type them, and then see the output from the shell (and any command you run under it).

Because we have disabled all of the normal input processing, the interrupt character (^*C*) becomes just another character. When your program reads a ^*C* (`0x03`) from the keyboard, it should use *kill(2)* to send a `SIGINT` to the shell process. Note that the shell will not necessarily die as a result of receiving this signal.

You may find it easier to tell if your program has recognized a ^*C* or ^*D* if, rather than echoing them directly back to the screen, you translate each into a standard graphical representation (echoing a caret in front of the letter: `^C` or `^D`).

You may find debugging complicated by the fact that it is not obvious which process did or did not receive or generate which character when. If you add a `--debug` option to your program that enables copious logging (e.g., to stderr) of every processed character and event, you may find that this greatly facilitates the debugging process.

4. shutdown processing:
   - Upon receiving an EOF (^*D*, or `0x04`) from the terminal, close the pipe to the shell, but continue processing input from the shell. We do this because there may still be output in transit from the shell.

     NOTE that a reader (e.g., your shell) will not get an end-of-file from a pipe until (all copies of) the write file descriptor (in all processes that share it) are closed. This means you have to close pipe file descriptors:

     a. in processes that don't use that end of that pipe.
     b. after your process is through writing to it (e.g., because you received an ^D)

The same is true of the pipe back from the shell to your process. Your *poll* will not return an error and your *read* will not return an EOF until the last write file descriptor associated with that pipe has been closed.

- Upon receiving EOF or polling-error from the shell, we know that (after we process the data already in the pipe) there will be no more output coming from the shell.
- Our first indication that the shell has shut down might also be the receipt of a `SIGPIPE` from a write to the pipe to the shell (i.e., because the shell has exited).
- After you have closed the write pipe to the shell and processed the final output returned from the shell, you should collect the shell's exit status (using *waitpid(2)* or a related function to await the process' completion and capture its return status) and report it to stderr in a line of the form:

  `SHELL EXIT SIGNAL=`# **`STATUS=`**#

  where the first # is the low order 7-bits (0x007f) of the shell's exit status and the second # is the next higher order byte (0xff00) of the shell's exit status (both decimal integers).

Note that the three normal shut-down scenarios (closing the pipe from the keyboard reader, receiving a `SIGPIPE` from the keyboard reader, and receiving an EOF from the shell reader) are not mutually independent. It is likely that, no matter how the shut-down is initiated, multiple of these events will occur in a non-deterministic order.

5. error checking

   Check for errors after all system calls. If system calls fail (for reasons other than shell shutdown) or arguments are unrecognized, print a meaningful error message to standard error, and exit with a return code of 1.

Summary of exit codes:

0 ... normal execution, shutdown on ^D
1 ... unrecognized argument or system call failure

SUBMISSION:

Your **README** file must include lines of the form:

NAME: *your name*
EMAIL: *your email*
ID: *your student ID*

And, if slip days are allowed on this project, and you want to use some, this too must be included in the **README** file:

SLIPDAYS: *#days*

If, for instance, you wanted to use two slip-days, you would add the following line:

SLIPDAYS: 2

Your name, student ID, and email address should also appear as comments at the top of your `Makefile` and each source file.

Your tarball should have a name of the form `lab1a-`*studentID*`.tar.gz`. You can sanity check your submission with this test script.

We will test it on a departmental Linux server. You would be well advised to test your submission on that platform before submitting it. Depending on the TA policy for your quarter, submissions that do not work on these servers may lose points.

A HISTORICAL NOTE ON *CR*, *LF*, AND *EOF*:

Long ago, when interaction was through mechanical teletypes with moving print-heads and paper rolls, these ASCII codes had the following meanings:

> 0x0D *Carriage Return* (or `'\r'` in C) meant move the printing head back to the left edge of the paper.
> 0x0A *Line Feed* (or `'\n'` in C) 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*> or `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 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).

GRADING:

Points for this project will be awarded:

value	feature
	Packaging and build (15% total)
5%	un-tars expected contents
5%	clean build w/default action (no warnings)
3%	Makefile has `clean` and `dist` targets
2%	reasonableness of `README` contents
	Basic Functionality (35% total)
5%	correctly detects/reports bad arguments
5%	correctly detects/reports system call failures
5%	correctly changes console to character-at-a-time, no-echo mode
5%	correctly restores terminal modes on exit
5%	keyboard input echoed one character at a time
5%	keyboard input written to screen one character at a time
5%	correct <*cr*> and <*lf*> handling
	--shell Functionality (50% total)
10%	forks a process and execs specified shell
10%	correctly passes keyboard input to screen or shell
10%	correctly passes output from shell to screen
5%	^D sends EOF to shell
5%	^C sends `SIGINT` to shell
5%	correct <*cr*> and <*lf*> handling

5% report shell exit status on exit