Programming Assignment 2

Due Thursday, September 17 before midnight

NAME

`slush` – The SLU shell

DESCRIPTION

`slush` is a very simple command-line interpreter. It uses a different syntax than shells like `bash`, and has much less functionality. `slush` executes in a loop in which it displays a prompt, reads in a command line from standard input, and executes the command.

There are two types of commands: built in commands which are executed by `slush` itself, and program execution commands which are carried out by separate processes.

A built in command must appear on a line by itself. The only built in command is:

`cd dir` – change current directory to dir

Program execution commands have the form:

$\texttt{prog}_n$ [args] (…$\texttt{prog}_3$ [args] ($\texttt{prog}_2$ [args] ($\texttt{prog}_1$ [args]

This command runs the programs $\texttt{prog}_n$, …, $\texttt{prog}_2$, $\texttt{prog}_1$ (each of which may have zero or more arguments) as separate processes in a“pipeline”. This means the standard output of each process is connected to the standard input of the next.

The syntax of `slush` is backwards from shells you're used to, and is intended to emphasize the functional nature of pipeline commands. As an example, the command line:

```
more ( sort ( ps aux
```

should produce a paginated, sorted list of processes.

`slush` should catch `^C` typed from the keyboard. If a command is running, this should interrupt the command. If the user is entering a line of input, `slush` should respond with a new prompt and a clean input line.

`slush` will exit when it reads an end-of-file on input.

ERROR HANDLING

If a program name does not exist or is not executable, `slush` should print an error message of the form:

```
prog1: Not found
```

(To do this automatically, you can use `perror()` after the `exec()` fails).

Syntax errors such as:

```
more ( ( ps au
```

```
more ( ps au (
```

should be handled with an appropriate message, such as:

```
Invalid null command
```

HINTS

Exiting on end-of-file (EOF) is VERY IMPORTANT because without it I cannot test your program. This is the only way to exit `slush`. You can create a EOF by typing `^D` at the keyboard, or by running your program with non-interactive input. For example, `echo ls | slush` should run `slush`, execute the ls command, and quit `slush`. I suggest you read input using `fgets()` in C or `cin.getline()` in C++. In both cases, the return value is false (0) on EOF.

To break the user input into tokens, you could use the C++ `stringstream`, or the C `strtok()` function. For simplicity, you may restrict each command in a pipeline to at most 15 arguments. You may also limit the length of an input line to 256 characters. Violations of these limits may result in an error or simply be ignored, so long as `slush` does not crash.

Turning the command into a pipeline of running processes is much easier if you write a recursive function. Start at the left end of the command string, and work right recursively. Processes should `fork()` off right to left as you return from the series of recursive calls.

When a process has a pipe, it needs to `close()` the file descriptor for the end of the pipe it's not going to use (or else it will never terminate). Do this after each `fork()`. Both parent and child may need to close something.

Count your children.. you'll need to `wait()` for them later.

Save the `^C` handling for last. `^C` generates a `SIGINT` signal. You need to set up a signal handler to catch the signal. What happens if a signal is caught while reading a line of input? How about if a signal is caught while `wait()` is waiting? One more hint for `^C` handling: typing `^|` will send a `SIGQUIT` and stop your process if you've screwed up `^C`.

USEFUL MAN PAGES

`strtok(3)`	`execvp(2)`
`strcmp(3)`	`fork(2)`
`chdir(2)`	`wait(2)`
`close(2)`	`signal(2)`
`dup2(2)`	`siginterrupt(3)`
`pipe(2)`	