Lab 7: Pipes in a shell

Objectives

- To gain experience in designing substantial modifications to a medium-sized C program.
- To become more familiar with the boundary between user programs and the Unix operating system.
- To understand more deeply how pipes operate.

Assignment

You can run *getcs* to fetch the C code for the shell that we studied in class. There are some minor modifications between this program and the textbook: First, the book's code has some function parameters that always match the global variables `envp` and `path`, and these have been cleaned out of this handout code. Second, the handout code places the `waitpid()` system call in the `executeCommand()` function, while the `main()` function accomplishes this in the book's code.

Your job in this lab is to modify the shell to support *pipes*, where one command's output can be fed into the input of another command. (We saw this in Lab 1.)

```
% cat mysh.c | grep open | wc
```

In this example command, we've told the shell that we want the output of the *cat* command to be fed as the input to the *grep* command, whose output should be sent to the *wc* command. The user would see only the output of *wc*.

To accomplish this, you will need the `pipe` system call in addition to other system calls and library functions seen in class.

`int pipe(int fd[2])`

> Allocates a pair of new, linked file descriptors, placing them into `fd[0]` and `fd[1]`. When any bytes are written to `fd[1]`, the operating system makes them available for reading from `fd[0]`. In other words, when you read from `fd[0]`, the operating system will return whatever has been written to `fd[1]`. The system call returns 0 on success and -1 if there is an error. (The most likely cause of an error is a lack of available file descriptors.)

Suppose the user enters the command "`ls | sort -r`". Your program will want to use the `pipe` system call to generate two file descriptors, fd_0 and fd_1. It will set up the *ls* process so that it writes to fd_1 and the *sort* process so that it reads from fd_0. This way, anything that the *ls* process outputs is sent as input to the *sort* process. The operating system handles this transfer of information between processes; the shell simply sets up the file descriptors to enable the transfer to take place, and it can then step back without intervening again.

The hint on the following page gives a detailed step-by-step approach to the problem.

Your solution should be able to work for any number of pipes in the same command. It should also allow for redirection into the first command or from the final command.

```
% cat < mysh.c | grep open | wc > count
```

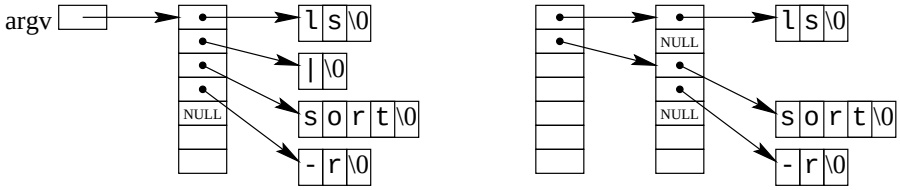
Since the shell already handles redirection, this should work once you get pipes working. But you should test your program to verify this.

Your lab report should contain your introduction and conclusion, along with the complete code *for the functions you altered only*, commented appropriately to reflect your modifications. Do not include code for functions you did not modify.

Run `handincs 210 7` to submit your code electronically.

Hint: The easiest way to accomplish this lab is to modify the `executeCommand()` function to follow a five-stage process. It isn't necessary to modify any other functions.

1. Break the command into its component parts by looking in the array of arguments for the string "`|`". Replace the pointer to each such word to a NULL pointer and create an array of pointers to the first argument of each component part. (Your program can assume that each "`|`" is a separate word when the user types the command.)



(a) before step 1 (b) after step 1

   Note that the new array is an array of `char**`'s.

2. Create an array to contain the standard input file descriptor for each part and another array for the standard output file descriptor for each part. If the command had three component parts, for example, the arrays might be as follows.

	index		
array	0	1	2
input descriptors	0	3	5
output descriptors	4	6	1

   Here, we suppose that `pipe` has been called twice; the first time, it placed 3 into fd_0 and 4 into fd_1, and the second time it placed 5 into fd_0 and 6 into fd_1.

   These two arrays can be interpreted as follows. The first column says that the first part of the command is to read from the keyboard (descriptor 0) and write to descriptor 4. The second part of the command is to read from descriptor 3 (which is linked to receive anything written to descriptor 4) and write to descriptor 6. And the final part of the command is to read from descriptor 5 (linked from descriptor 6) and write to the screen (descriptor 1).

3. Now execute each component part of the command separately, without waiting for each part to complete. Your shell will be creating several children running simultaneously. Save the process ID for the final child (as returned by `executeExecutable`) in a variable for the purpose of step 5 below.

4. Close all the file descriptors in the arrays (except for 0 and 1). This is important because these file descriptors are currently allocated by the shell, and we don't want them eating up room in the limited file descriptor table. (Note that closing a file descriptor in one process doesn't harm the descriptor for any child processes who inherited the descriptor. That's important, since the children may still be using the pipes that the parent process is now closing.)

5. Wait for the final part of the command to complete, using the `waitpid` system call.