Information about shell project

I. Introduction

For this project you will implement a simplified shell.
We have provided a command parser that reads a command
provided at the command prompt and creates a tree for you.
Your assignment is to process that tree so the command
represented by the tree is executed. The function you
need to implement can be found in the executor.c file:

```
int execute(struct tree *t) {
}
```

Although you need to provide a makefile for the project
you can start implementing your code without it. To see
what the parser does so far, compile the project code
distribution as follows:

```
gcc *.c -lreadline
```

Once you compile, execute a.out. You wil see the shell
prompt (d8sh%). If you type any command and press enter,
you will see the message "You must implement me :)".
To stop the shell (which as you can see is a C program)
press CTRL-C for now. Edit the execute function above
so the function calls print_tree(t) to print the
tree that represents the command that needs to be
executed. After adding the print_tree function call,
compile the code and execute a.out. Now type the ls command
at the d8sh% prompt. You will see the following:

```
NONE: ls, IR: (null), OR: (null)
```

The command tree is a binary tree where a C structure
represents a tree node (see the file command.h). Each
node has a type. For this project you will implement
commands associated with the NONE, AND, PIPE and SUBSHELL
nodes. Each node has a left and right subtree (which are
null if not subtree is present). In addition, there is an
each node has an argv array that has the command arguments.
The input field is a string that represents a file used
for input redirection and the output field represents

the file used for output redirection. At this point,
execute each of the commands below at the d8sh% prompt
and look at the output. This will help you understand how
the fields of a NONE node (also known as NONE conjunction)
are populated with data. For example, t->argv[0] has the
command to execute, and the rest of t->argv will have command
line arguments. t->input is the file used for input redirection;
t->output the file used for output redirection.

Commands to execute at the d8sh% prompt:
---

a.out
a.out < data.txt
a.out > results.txt
a.out < data.txt > results.txt
a.out arg1 arg2
cd
exit
cd /tmp

A NONE node is just a leaf of the tree. The NONE conjunction
is the one you need to implement first. It is the one
associated with cd, exit, commands with arguments (e.g.,
a.out arg1 arg2) and commands that use input and output
redirection. This part is similar to Shell Jr. Shell commands
(cd, exit) don't require fork/exec; linux commands will require
fork and the child will perform an exec (execvp(t->argv[0], t->argv);)
to execute the command. BEFORE the child performs the execvp call,
the child needs to handle input/output redirection using dup2.
If t->input is different than NULL, then open the file
t->input and use dup2 to change the standard input of the child
to use that file. If t->output is different than NULL, then open
the file t->output and use dup2 to change the standard output
of the child to that file.

Remember that in linux, a successful command execution is
represented by a program returning a value of 0; any other
value indicates the command failed for some reason. In tcsh
you can find out the status of the last executed command
by using echo $?. The following is an example of executing
a valid and invalid command in the grace cluster and the
value printed after the command execution.

```
% date
Fri Jul 23 14:37:52 EDT 2021
% echo $?
0
% cat bla
cat: bla: No such file or directory
% echo $?
1
%
```

The execute function (int execute(struct tree *t))
you need to implement needs to return 0 to indicate
successful execution of the command represented
by tree and a non-zero value otherwise.

II. Processing commands that involve more than one node

If you have a command that is just a NONE node (a root node
and nothing else) you will be done after implementing the code
for the NONE none, but commands can be more complex.
In a linux shell we can use the following options:

1. AND && (two &)

   You can connect two or more linux command with &&.
   Once a command fails, the rest are not executed.
   For example executing in the grace cluster (assuming
   the file lexer.h is present) will allow the following date
   command to execute:

   ```
   % ls lexer.h && date
   lexer.h
   Fri Jul 23 14:54:11 EDT 2021
   %
   ```

   If we try the following command, where the file does not
   exist, the date command will not be executed.

   ```
   % ls lexer.h && date
   lexer.h
   Fri Jul 23 14:54:11 EDT 2021
   %
   ```

Your shell will process &&. Execute in the d8sh% the following command:

a.out && a.out2

You will see the following output that is a tree with 3 nodes (an AND node (root) and two leaf nodes).

NONE: a.out, IR: (null), OR: (null)
&&, IR: (null), OR: (null)
NONE: a.out2, IR: (null), OR: (null)

2. PIPE | (single |)

A pipe allow us to connect the output of one process to the input of another. For example, the wc command in linux provides information about the number of lines, words in a file. Here is an example:

```
% cat info.txt
This is
a cat
and a cat
is a nice cat
% wc info.txt
4 11 38 info.txt
%
```

What if you would like to count lines and words associated with the output of ls? You can use a pipe where the output of ls will be fed to wc as follows:

```
% ls | wc
  13      13     145
%
```

You can add more pipes (e.g., the output of wc can be used by another command).

Your shell will process |. Execute in the d8sh% the following command:

a.out | a.out2

You will see the following output that is a tree with 3 nodes
(a PIPE node (root) and two leaf nodes).

NONE: a.out, IR: (null), OR: (null)
|, IR: (null), OR: (null)
NONE: a.out2, IR: (null), OR: (null)

To process the PIPE you will need to create a pipe
(as shown in lecture) and pass the appropriate file
descriptors to the children.

As you can see, the tree can be large and can have
a combination of PIPE and/or AND nodes. AT THE LEAF
LEVEL OF THE TREE YOU WILL FIND THE NONE nodes.

III. Processing the tree
==========

At first, it looks like processing the tree is difficult due
to the different types of nodes, the input/output redirection,
and the different types of commands, but that is not the case.
The tree has been created for you, what simplies matters a lot!
You can implement the project with simple commands that use
&& and | and input and output redirection. IMPORTANT: The impact of
the PIPE and AND nodes in the tree is to define which file
descriptors leaf nodes (NONE nodes) will use for data input
and for data output.

To process the tree you need to start at the root, processing
each node. Each tree node will receive from its parent node
an input file descriptor (integer) that represents what the
node should use as the source of input (e.g., a file,
standard input, a pipe read end) and what should use for
output (e.g., a file, standard output, a pipe write end).
The processing that takes place at each node will depend
on the kind of node, the file descriptors it receives from the
parent, and whether the node has files for input/output redirection
(i.e., whether t->input and t->output are different than NULL).
For example, a PIPE node will create a pipe and then decide
what input and output file descriptors need to send to its
children. The AND node will just pass the file descriptors
it receives to its children. How can the file descriptors
be passed to children? There are different approaches, but

one approach is to define an auxiliary function that we
can call recursively to process the tree. This function
could be:

```
static int execute_aux(struct tree *t, int p_input_fd, int p_output_fd) {
}
```

The function receives a tree, a parent input file descriptor
(p_input_fd) and a parent output file descriptor (p_output_fd).

The execute function will call the above auxiliary as follows:

```
int execute(struct tree *t) {
  return execute_aux(t, STDIN_FILENO, STDOUT_FILENO);
}
```

As you can see the root node will receive standard input as the input file
descriptor and standard output as the output file descriptor, as
the root has no parent.

Every node (NONE, AND, PIPE, SUBSHELL) relies on four pieces of
information to make a decision regarding that file descriptors
it will pass to its children. Assuming t points to a node,
the four pieces of information are:

a. t->input (file provided for input redirection)
b. t->output (file provided for output redirection)
c. Input file descriptor provided by the parent node (p_input_fd)
d. Output file descriptor provided by the parent node (p_output_fd)

The processing for each kind of node is:

1. NONE node
==========

If the command is exit or cd, the shell can process it directly without
doing any fork/exec/dup2. For a command other than exit or cd
you need to:

a. Fork so the child processes the command and the shell waits
for the command to be executed by the child. The shell will
return 0 if the child processed the command successfully and another
value otherwise (any value should be fine). This value is what
we refer to as the status value (the integer value returned

by the execute function and the execute_aux() function).

b. In the child, BEFORE we execute execvp, we need to decide what the program we are executing will use for input/output. Does it relies on the files it may have for input/output redirection?

For a NONE node you will check whether the command relies on input/output redirection. If t->input is present (value other than NULL), you will ignore the input file descriptor provided by the parent (p_input_fd) and use the file descriptor associated with opening the file t->input. If t->input is NULL, use the file descriptor provided by the parent (p_input_fd). A similar processing applies to t->output. After deciding which file descriptors you need to use, USE dup2 TO MAP the file descriptor 0 of the child process to the file/pipe end that will be used for input, and file descriptor 1 of the child process to the file/pipe end that will be used for output. Once this process has taken place, then execvp can be called and the program will execute using the correct data source/data destination. IF YOU execute dup2 BEFORE YOU FORK, YOU WILL MESS UP THE FILE DESCRIPTORS OF THE SHELL PROCESS; THAT IS WRONG.

2. AND node
==========

a. Process the left subtree (t->left) using the parent input/output file descriptors (p_input_fd, p_output_fd) that the node received.

b. If the left subtree execution is successful, the right subtree (t->right) will be processed using the parent's input/output file descriptors (p_input_fd, p_output_fd). In this case the value returned by the AND node processing will be the status value returned by processing the right subtree.

If the left subtree execution failed, then the right subtree will not be processed and the AND node processing will return the status value returned by the processing of the left subtree.

3. PIPE node
==========

a. Create a pipe

b. One process will take care of left subtree and another of the right tree; which one is up to you.

c. For the left subtree

If t->input is defined, open the file and use it as input file descriptor for the left subtree; otherwise use what the parent provides (p_input_fd). The output file descriptor will be the write end of the pipe.

d. For the right tree

If t->output is defined, open the file and use it as output file descriptor for the right subtree; otherwise use what the parent provides (p_output_fd). The input file descriptor will be the read end of the pipe.

4. SUBSHELL node
==========

Note: A subshell is a child process launched by a shell. Parentheses start a subshell. Commands included in the parentheses will not affect the parent shell's environment (e.g., the working directory). For example, executing:

```
( cd /tmp && ls )
```

will list the files of the directory /tmp, but the working directory after the above command has been executed will not be /tmp. If you remove the parenthesis, the working directory will be /tmp. You can determine the working directory by executing the pwd command.

a. The processing of a subshell will be done by a child process (i.e., you need to fork()) that processes the t->left subtree of the SUBSHELL node. WARNING: You will ignore the t->right subtree.

b. If t->input is defined, open the file and use the file descriptor as the input file descriptor to process the t->left tree. Notice that the original shell received standard input/output; you are doing a similar process, but this time what you are passing to the subshell is either the file descriptor received from the parent or the one associated with t->input (if t->input is different than NULL).

c. A process similar to the one described in b. applies to t->output.

d. The status value to return will be status of processing the
left subtree.

e. The parent will reap the child that is executing the subshell.

IV. Miscellanenous

1. WARNING: Make sure your code compiles without warnings, otherwise you will lose credit. For example, make sure you comment out the print_tree function provided with executor.c once you have no longer use it. If you do not comment it out, you will lose credit.

2. WARNING: Make sure your executor.c file has your student id at the top of the file, otherwise you will lose credit.

3. WARNING. A dup2 system call should only happen after a fork(). A common error is to perform a dup2 before a fork, what changes the shell standard input/output file descriptors instead of standard input/ouput file descriptors of the child process. The command is executed, but the shell now has its file descriptors corrupted.

4. What is ambigous input redirect and ambigous output redirect??

   Here are examples:

   Ambiguous output redirect:

   ```
   a.out > result | gone
   ```

   Ambiguous input redirect:

   ```
   a.out | gone < data
   ```

   You need to recognize the above cases and stop processing and return a value other than 0. The output messages to use are:

   ```
   printf("Ambiguous output redirect.\n");

   printf("Ambiguous input redirect.\n");
   ```

   Once you recognized the PIPE conjunction, but before you do any

processing, verify whether you have ambiguous output redirect
and ambiguous input redirect. First check for ambiguous output
redirect; if you identify this condition, stop processing the
pipe conjunction. If there is no ambiguous output redirect,
check for ambiguous input redirect and stop processing the
pipe conjunction if you identify this condition.

No processing should be perform if ambiguous redirects are identified.

IMPORTANT IMPORTANT
Make sure you use the printf statements we left in these notes
as students often fail tests due to typos in the expected messages.

5. WARNING: If you are adding printf statements for debugging, make
sure you use fflush(stdout) so your output is guaranteed to take place.

6. In this project project we don't have background processing. The
loop we use to reap (in one of our lecture examples) is not needed
for this project.

7. WARNING: Remember, that you create a pipe and THEN fork (otherwise
the pipe will not be shared). Remember: PFChang (Pipe and Fork)

V. Videos
=========

1. The following video covers some aspects of the shell project:

https://umd.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=10c89d23-a806-454d-955c-aa940129a742

There is a correction that needs to be made:

In the video it is mentioned that for the && conjunction you check the t->input and t->output
and open files and pass file descriptors, but for this version of the shell this is not necessary.
The && conjunction will just use the file descriptors it receives from the parent
and pass them to the children. The left child will be executed first and if successful
the right child will be executed. There is no need to fork while processing the actual
&& conjunction.

2. The beginning of the following video goes into detail on how to implement subshell:

https://umd.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=de9726ca-014d-46ea-824e-aa9601296f22#