Project 2 run through

Jesus Nuñez
Colorado School of Mines

January 30, 2022

The Datastructure

```c
struct command {
    /* The collected arguments. */
    char *argv[ARGS_MAX];

    /*
     * The input file, or NULL if none.
     * Note: it is not valid to have an input file on the
     * receiving end of a pipe.
     */
    char *input_filename;

    /*
     * The output type.
     *
     * COMMAND_OUTPUT_STDOUT:
     *     Output to the terminal.
     * COMMAND_OUTPUT_FILE_TRUNCATE:
     *     Output to the file specified by output_filename.
     *     Overwrite any existing contents.
     * COMMAND_OUTPUT_FILE_APPEND:
     *     Output to the file specified by output_filename.
     *     Append to any existing contents.
     * COMMAND_OUTPUT_PIPE:
     *     Forward the output to the input of the command
     *     specified by pipe_to.
     */
    enum command_output_type output_type;
    union {
        /* When COMMAND_OUTPUT_FILE_*, this is set. */
        char *output_filename;
        /* When COMMAND_OUTPUT_PIPE, this is set. */
        struct command *pipe_to;
    };
};
```

We will look at these variables one at a time

argv

```
argv     0      1      2    3    4    5
       echo hello  this  is   a command

argv     0      1          2
         ls     -l     folderName
```

This is how two different Linux commands will look like in argv.

The zero element will always be the command being ran and the following elements are the arguments the command is called with.

The argv array is what you need to pass into whichever exec function you decide to use. The exact way to pass it in is described in the man page for the exec family of functions.

input_filename

```
cat < inputfile.txt
```

- The variable is NULL if there is no $<$ character.
- If there is a $<$ and a file is provided, then the variable is a char array of that filename.
  - In the example provided in the image, input_filename will be "inputfile.txt"
- You must open the file and dup the file descriptor to standard in. This will make the contents of the file automatically go to cat.

output_type

```
echo hello

echo hello > truncate_output.txt

echo hello >> append_output.txt

echo hello | grep hello
```

There are 4 main types of output.

1. standard output where it prints to terminal
2. truncation to file where the contents of the file are deleted and replaced with the output of the command before the $>$
3. append to file where the output of the command is put at the end of the filename
4. pipe to another command where the output of the command on the left of the $|$ is turned into the input of the command on the right

output_type

```
echo hello

echo hello > truncate_output.txt

echo hello >> append_output.txt

echo hello | grep hello
```

To identify which needs to happen we have an enum declared.
This enum is what is set in this output_type variable.
The enums go as follows.

- COMMAND_OUTPUT_STDOUT
- COMMAND_OUTPUT_FILE_TRUNCATE
- COMMAND_OUTPUT_FILE_APPEND
- COMMAND_OUTPUT_PIPE

output_filename

```
echo hello > truncate_output.txt

echo hello >> append_output.txt
```

Much like input_file, this variable is NULL if there is no > or >>

This option is a char array if and only if output_type is
COMMAND_OUTPUT_FILE_TRUNCATE
or
COMMAND_OUTPUT_FILE_APPEND

output_filename

```
echo hello > truncate_output.txt

echo hello >> append_output.txt
```

Remember Truncate means you need to erase everything in the file and replace it with the new contents.

Append means you need to add the command output to the bottom of the file (if it already exists)

If the file doesn't exist you create it.

pipe_to

```
echo hello | grep hello

echo hello | cat | cat

echo hello | cat | cat | cat
```

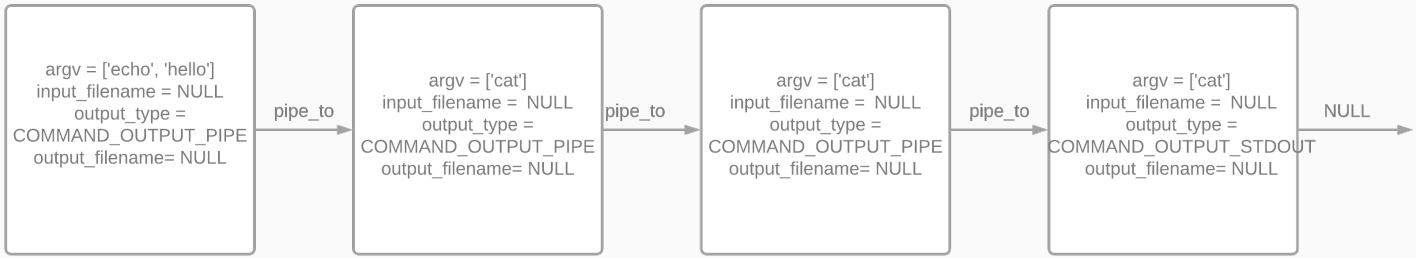
pipe_to much like all the file input and output is NULL if there is no | character.
The data structure is a type of linked list. The first command is in the first struct and the next command is in the struct that pipe_to points to.

pipe_to

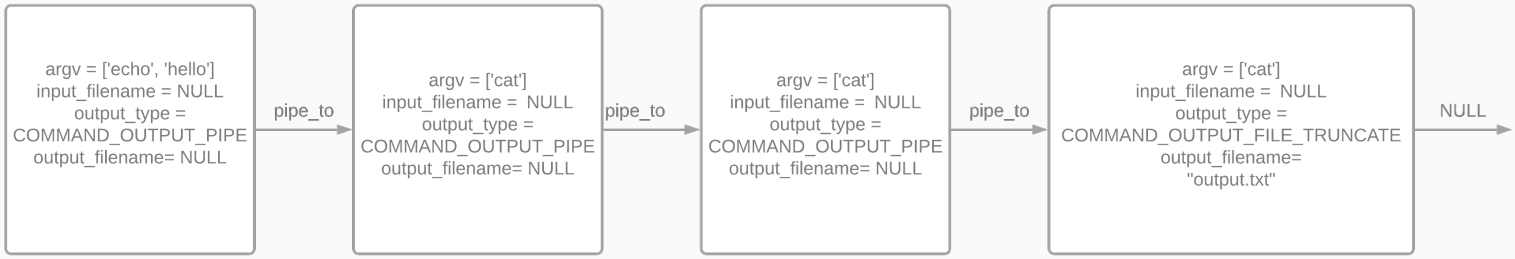
This slide is here purely to say that the next slide is a picture that shows how the struct is made for commands.
A command can have both an input file and an output file. It will just look like a combination of the two shown.

pipe_to

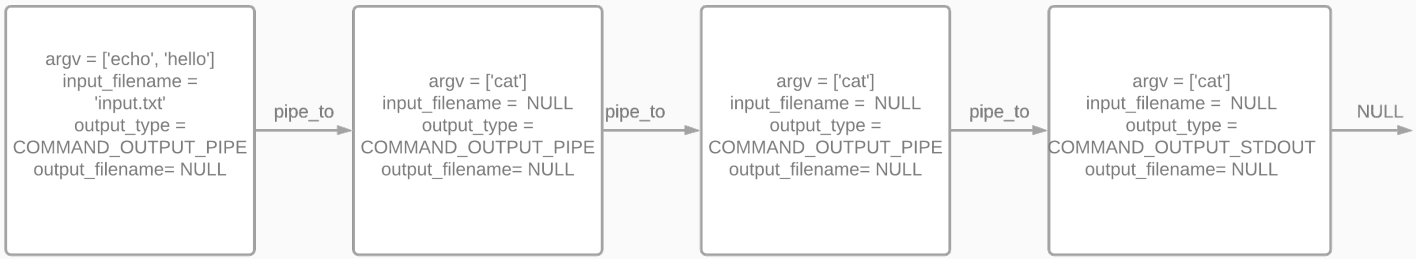
echo hello | cat | cat | cat



echo hello | cat | cat | cat > output.txt



cat < input.txt | cat | cat | cat



Invalid commands

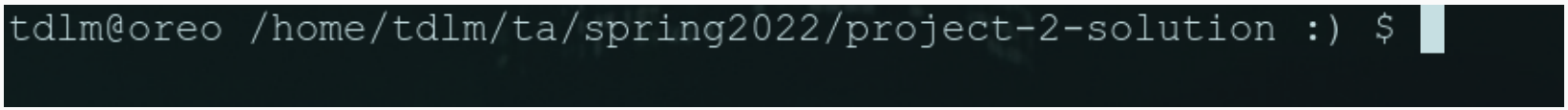
```
echo hello | cat < input.txt | cat

echo hello | cat > output.txt | cat | cat
```

Input will ALWAYS be on the first command never in the pipe as seen in the first example.
Output will ALWAYS be on the end and never in the middle of the pipe

The project

When you run your shell you will get something that looks like this. Note the username, host and the directory will be different.



You run your commands here like a normal terminal. Do note the smile. The smile means that the command ran successfully.

Project Details

```
tdlm@oreo /home/tdlm/ta/spring2022/project-2-solution :) $ echo hello
hello
tdlm@oreo /home/tdlm/ta/spring2022/project-2-solution :) $
```

In this example we see that the command 'echo hello' resulted in a smile. Echo ran correctly and it created a smile on the new terminal line.

```
tdlm@oreo /home/tdlm/ta/spring2022/project-2-solution :) $ grep
Usage: grep [OPTION]... PATTERNS [FILE]...
Try 'grep --help' for more information.
tdlm@oreo /home/tdlm/ta/spring2022/project-2-solution :( $
```

In this image we see that the grep command ran BUT it gave an error. This resulted in a frown in the next line even though the command was real.

Starting the project

All the work you will be doing is inside a file called dispatcher.c

This file has a function that you need to implement called dispatch_external_command. This function is incharge of running the command given. It takes the struct described in the previous slides as input.

The parser, terminal, and all other code is already implemented.

Recommendation/hints

- Split what you need to do into functions
  - Having functions that correctly dup input and output for you depending on if the output_type is a pipe, truncate, or append is really helpful (keeps that logic in one spot so you don't have to edit it in 5 places)
- Make sure that you exit any child process. If you don't you will end up with a shell running inside a shell.
  - When you run your code typing 'exit' once should exit your code. If it takes more than this you forgot to exit from the child

Recommendation/hints

- As for implementations for handling the logic. There are two methods you can do.
  - Recursive or a Loop
  - You can recurse through the list of commands and run them one at a time redirecting output as needed
- The order that the commands are run in does not matter. A command will wait for input before running.
  - 'echo hello | grep h | cat | less'
  - Let's say that in your implementation you run less first and then run the commands backwards.
  - Less won't run until it recieves input so it will wait for cat to run.
  - In turn cat will wait for grep and so on.

Important things to check

```
tdlm@oreo /home/tdlm/ta/spring2022/project-2-solution :) $ echo hello | false | cat
tdlm@oreo /home/tdlm/ta/spring2022/project-2-solution :) $
```

In Linux, false is a command that just fails. So in this command the first command in the pipe (echo) passes successfully then false fails but this then goes into cat which passes. This ultimately results in a smile because the exit code of the last command is what matters.

```
tdlm@oreo /home/tdlm/ta/spring2022/project-2-solution :) $ echo hello | cat | false
tdlm@oreo /home/tdlm/ta/spring2022/project-2-solution :( $
```

In this command, the failure is at the end thus it does result in a frown.

Copyright Notice