50.005 CSE

INFORMATION SYSTEMS TECHNOLOGY AND DESIGN

Lab 4: File Operations

Natalie Agus

1 Overview: File Operations

There are numerous file operations that we can find in Shell, such as display file content with `cat`, list files in current directory with `ls`, etc. In this lab, we are going to manually implement some file operations in our custom shell interface.

2 The Starter Code

Download the starter code, `shell.c` and `shell.h`. Take a look at the codes to to get its general idea:

1. Go to `shellPrograms` folder inside the starter code. Open terminal, cd to this location and type `make`. This recompiles the .c files into executables.

2. `shell.c` and `shell.h` contains a simple example of custom shell interface

3. When you compile and run the code, you will see the following,

```
Natalies-MBP-2:Lab4 natalie_agus$ gcc -o shell.o shell.c
Natalies-MBP-2:Lab4 natalie_agus$ ./shell.o
CSEShell>
```

Figure 1

   Tired of typing that `gcc -o shell.o shell.c` everytime you want to recompile? Try making your own `makefile` instead. Refer to the `makefile` in Lab 1.

4. Try typing `help` and press enter, you will see a list of commands that are supposedly implemented in the interface.

5. Right now we have 9 built-in commands, 3 of which have been implemented for you: `cd`, `help`, and `exit`:
   (a) `cd`: changes the working directory to another
   (b) `help`: list all the builtin commands implemented
   (c) `exit`: exits the custom shell interface
6. For this lab, you are tasked to implement the other 5 functions **[30m]**, 1 bonus function and 1 bonus option.
7. You can try typing other commands: `find yourfilename, listdir, listdirall, countline yourfilename, display yourfilename`, where `yourfilename` is your target file, e.g: `1.txt`. You will be greeted with hello message for each command. You are supposed to replace these hello message with your code that implements the requirements of each command.
8. Note that each time you type in a command, the custom shell executes your command and give you a new prompt line.
9. **Warning**: after you `cd`, you will encounter error if you try the custom commands `find yourfilename, listdir, listdirall, countline yourfilename, display yourfilename` because the path to the executables in the `execvp` functions are no longer correct. To make these commands still work even after you `cd`, change the paths into absolute paths to the respective executables in your machine. *It is not necessary to do so to complete this lab. Just do not `cd` if you want to try any of the five custom commands above.*

2.1 How the Interface Works in a Nutshell

1. When you run the code, the function `shellLoop()` will be executed from the `main` function
2. `shellLoop()` will run indefinitely as long as `status` is not zero
3. In `shellLoop()`, we read line from stdin through function `shellReadLine`, tokenize it based on several delimiters defined in `shell.h` through function `shellTokenizeInput`, and execute the command through function `shellExecuteInput`.
4. How do we execute the command? Have a look at this line of code inside `shellExecuteInput`:

```
// Check if the commands exist in the command list
for (i = 0; i < numOfBuiltinFunctions(); i++) {
  if (strcmp(args[0], builtin_commands[i]) == 0) {
      if (i != 0 && i!=1 && i!=2){
          /**create new process to run the function with
          the specific command, Except for cd, help, and exit.
```

```
                These three have  to be done in  this (current)
                process space**/

                pid = fork();

                if (pid == 0){
                    int status = (*builtin_commandFunc[i])(args);
                    exit(status);
                }
                else if (pid < 0)
                {
                    perror("Fork doesn't work, exiting Program.");
                }
                else{
                    //wait until process has finished running

                    waitpid(pid, &status, WUNTRACED);
                    return status;
                }
            }
            else{
                return (*builtin_commandFunc[i])(args);
            }

        }
    }
```

It basically loops over the list of commands we have. If it matches the currently implemented list of commands, and if the commands are not `cd`, `help`, or `exit`, it will create a new process using `fork()`, and run that command using : `int status = (*builtin_commandFunc[i])(args)`, and wait for the process to finish using `waitpid(pid, &status, WUNTRACED)`. Otherwise, an error message like the following will show:

```
CSEShell> somerandomfunction
Invalid command received. Type help to see what commands are implemented.
CSEShell>
```

Figure 2

2.2 About `execvp`

In `shell.c`, take a look at each of the functions you are suppose to implement: `shellDisplayFile`, `shellCountLine`, `shellListDir`, `shellListDirAll`, and `shellFind`. Notice that all of them calls `execvp`. If you go online and find out what it is supposed

to do, you will see that `int execvp(const char *file, char *const argv[])` will **execute what is specified in the path** `const char* file`**, with argument** `argv[]`. More importantly, **execvp replaces the current process image with the new process image of the target executable** `file`. Therefore, whatever lines of code underneath `execvp` **will not be executed** unless `execvp` fails to launch.

Hence, **this is why it is important to** `fork()` a new process before calling `execvp`. Otherwise, your entire custom shell program will exit as soon as you `execvp`. If you want to try, you can replace the entire for-loop inside `shellExecuteInput` function shown in (4) above into:

```
// Check if the commands exist in the command list
for (i = 0; i < numOfBuiltinFunctions(); i++) {
  if (strcmp(args[0], builtin_commands[i]) == 0) {
   return (*builtin_commandFunc[i])(args);
  }
}
```

Compile `shell.c` using command `gcc` as shown in Figure 1, run the output, and type `listdir`. You will be greeted with a hello message and your shell will exit, instead of displaying a new prompt line.

2.3 Your Custom Command Programs

Open the folder `shellPrograms`. Notice that you will have 5 .c files, each implements the requirements of each commands, one .h file that serves as the header file for all .c files, a makefile to compile them conveniently, and five **executables** called `countline, display, find, listdir, listdirall`. Try double clicking on them. A new terminal window will launch and you will be greeted by welcome messages. Double clicking them basically running the `main` function in it without any arguments.

The script `shell.c` basically calls each of these executables using `execvp` in its respective functions, with the appropriate arguments.

3 Your Tasks

Your job now is to implement each of the commands in each of the five .c files. Remember to call `make` using your terminal (after you change directory to `shellPrograms`) to recompile the codes and produce new executables.

3.1 [5m] Implement `shellDisplayFile_code`

Implement the function `shellDisplayFile_code` inside `shellDisplayFile_code.c` that will display the contents of a file using : `display filename`. It is similar to how `cat` works in UNIX-based terminals, as shown in the figure below:

```
CSEShell> display loremIpsum.txt
Lorem Ipsum is simply dummy text of the printing and typesetting industry.
Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, wh
en an unknown printer took a galley of type and scrambled it to make a type spec
imen book.
It has survived not only five centuries, but also the leap into electronic types
etting, remaining essentially unchanged.
It was popularised in the 1960s with the release of Letraset sheets containing L
orem Ipsum passages, and more recently with desktop publishing software like Ald
us PageMaker including versions of Lorem Ipsum.
CSEShell>
```

Figure 3

These c-functions might help you in your implementation: `fopen, fgets, fclose`.

3.2 [5m] Implement `shellCountLine_code`

Implement the function `shellCountLine_code` inside `shellCountLine_code.c` that will display the number of lines in a file using : `countline filename`, as shown in the figure below:

```
CSEShell> countline loremIpsum.txt
There are 4 lines in loremIpsum.txt
CSEShell>
```

Figure 4

A line is defined as a series of characters that are terminated by `\n` character. This function terminates once you meet `EOF` (end-of-file) character, which means that all the contents of the file has been read. These c-functions might help you in your implementation: `fopen, fgets, fclose, getc`.

3.3 [5m] Implement `shellListDir_code`

Implement the function `shellListDir_code` inside `shellListDir_code.c` that will display all the names of the files under the current directory using the command : `listdir`. Your output should look like the screenshot below:

```
CSEShell> listdir
.
..
2018_Lab4_handout.pdf
shell.h
shell.o
.DS_Store
s.o
combined.txt
fileOps.h
fileOps.o
05-syscalls.pdf
program.bin
Handout
shell.c
loremIpsum.txt
writeSentence.txt
2.txt
1.txt
merge.txt
hello.txt
fileOps.c
CSEShell>
```

Figure 5

These c-functions/struct might help you in your implementation: `strtok`, `opendir`, `readdir`, `struct dirent`.

3.4 [10m] Implement `shellListDirAll_code`

Implement the function `shellListDirAll_code` inside `shellListDir_code.c` that will display all the names of the files under the current directory and subdirectories using the command : `listdirall`. It produces similar output format as `shellListDir`. You might want to implement a recursive function to traverse all the subdirectories. These c-functions/struct might help you in your implementation: `strtok`, `opendir`, `readdir`, `struct dirent`, `sprintf`, `strcmp`.

3.5 [5m] Implement `shellFind_code`

Implement the function `shellFind_code` inside `shellFind_code.c` that will display all the names of the files under the current directory and subdirectories that matches the input filename using the command : `find filename_keyword`. Your output should look like the screenshot below:

```
CSEShell> find .txt
./combined.txt
./loremIpsum.txt
./writeSentence.txt
./2.txt
./1.txt
./merge.txt
./hello.txt
CSEShell>
```

Figure 6

These c-functions/struct might help you in your implementation: `strtok`, `opendir`, `readdir`, `struct dirent`, `sprintf`, `strcmp`. A quick way is to reuse `shellListDir` to help you traverse through the directories, and print only those files that matches `filename`.

3.6 [10m] Bonus: Implement `shellUsage` function (inside `shell.c`) and implement options for `shellListDir_code`

(1)**[3m]** implement the function `shellUsage` that prints out how to use all the implemented commands, so for example, typing `usage cd` will print a guide on how to use the command `cd` properly: `Type: cd directory_name`. Your output should look like the screenshot below:

```
CSEShell> usage help
Type: help
CSEShell> usage find
Type: find filename_keyword
CSEShell> usage exit
Type: exit
CSEShell> usage listdir
Type: listdir
Type: listdir -a to list all contents in the current dir and its subdirs
CSEShell>
```

Figure 7

Notice in the screenshot of `usage` above, that `listdir` can take an option `-a` to turn it into `listdirall`. (2) **[2m]** Add the option `-a` in `shellListDir_code` function to do this. In other words, if one types `listdir -a`, the shell should return you all the filename under the current directory as well as its subdirectories, which is exactly what `listdirall` will do. *Hint: you will need to `execvp listdirall` executable from `shellListDir_code` if theres -a option.*

This `-options` feature common in bash commands, for example, when we compile, we use `gcc -o shell.out shell.c`, where `-o` represents the option for output.

(3) **5m** Finally, **implement your own custom file operation function, and you must use** `execvp`. You can do anything pertaining to file operation, such as rename,

delete, copy, etc. Create a new .c file and you are free to implement anything. Do not forget to add this function in `shellPrograms.h`. You can modify the `makefile` to ensure that you produce the the executable using `make` and not having to compile it separately. Do not forget also to modify `shell.c` and `shell.h` to add this new command. Write a short `readme.txt` file to tell us what command you have implemented and its function.

4 Submission

The maximum marks for this lab is capped at 30, so you can score full marks without even doing the bonus part. Scoring the bonus section after getting full marks will not give additional marks. **The lab can be done in a groups of 3 at maximum, so you can choose to work solo or in pairs as well**. Zip all the files in `starterCode` folder, together **with a readme file stating your group member names and student id**. Only **ONE** person per group has to submit.