CS 131a: Operating Systems

Programming Assignment 1

Released: 9/20/2021
Date, room and Zoom link for highly encourage tutorial will be announced on LATTE)
Due: 10/1/2021

Getting Started and note on the Future

Your assignment is going to be to build a basic command line environment (shell) for interacting with a file system. This command line environment will be written in Java. One of the goals of this assignment is to refresh your coding ability in Java and ensure that you have the coding pre-requisites to be able to complete this course. You will also learn about Unix shell commands and will get to practice using Java-based System calls. An important difference between the shell commands you will be implementing in this assignment and the commands in a real shell is that your commands will run sequentially. That is, one command terminates before the next one starts. In contrast, real shell commands can run concurrently. E.g. a command that searches a file A for a certain phrase, and another command that writes the found phrases to file B can proceed in parallel, thus potentially completing the work faster. But no worries, you will learn to build such commands too! The other goal of this assignment is to develop the groundwork for your Programming assignment 2 that will "teach" your shell commands to run concurrently. Since your Programming assignment 2 will rely on the code you develop in this assignment, to make this job easier, we will ask you to follow strict instructions how to structure your code in this assignment. And now off we go!

Understanding UNIX shell

Essential to this assignment is understanding the fundamentals of a UNIX shell, and the functionality that is expected from a UNIX like operating system. The following pages can help you get a sufficient understanding of the expected behavior of UNIX and shell commands to allow you to complete this project. After reading through these pages, it is highly recommended that you play around on a UNIX console before you attempt to construct your own:

Information	What you need to know	Link
What is Unix?	Theoretical	https://bit.ly/3pVGXb5
Working Directory Commands	ls, cd, . , .. , pwd	https://bit.ly/3rsxJmP
Basic Text Operations	cat, grep, wc, uniq, head, tail	https://bit.ly/36LSQJ6
Redirection and Piping	Redirection, Piping, > |	https://bit.ly/3rsxOa7

Your assignment: Implementing the REPL Loop

Your main program will include a loop that prints a simple prompt (“>”), accepts a command, and prints the output from that command (if any). This is called a ***filter*** (e.g., `cat hello-world.txt` is a filter). Your program should also handle piped filters, i.e., multiple commands separated by pipes (e.g., `cat hello-world.txt | grep world`). This type of loop is sometimes called a Read-Eval-Print Loop, or REPL. The REPL loop should only exit upon user request. For example:

```
> Welcome to the Unix-ish command line.
> cat hello-world.txt
hello
world!
> not-a-command
The command [not-a-command] was not recognized.
> cat hello-world.txt | grep world
world!
> cat hello-world.txt | grep world > output.txt
> cat output.txt
world!
> ls
src
hello-world.txt
output.txt
> exit
Thank you for using the Unix-ish command line. Goodbye!
```

Commands

Your program should correctly parse commands in a way that allows the creation of piped filters. Your REPL loop also must support simple directory commands: `ls`, `cd`, and `pwd`. These commands are a bit different, as `cd` does not take piped input and does not produce output and `ls` and `pwd` create piped output, even though they don't need piped input (the same is true for the command `cat`). Your program is expected to implement the following Unix commands. Read the descriptions carefully, we have simplified some commands, to ease your implementation task. A proper implementation of a command therefore, should refer back to the correct error messages to print, and the behavior as specified below.

Command	Piped input	Piped output	Short Description	Notes
File System Navigation Commands				
Pwd	No	Yes	Pipes the working directory to the output message queue	You may want to use the Java File class (also see the slide about the File class in the slides from the first tutorial). You can also use System calls to achieve this.
ls	No	Yes	Pipes the contents of the current working directory to the output message queue	You do not need to allow arguments to ls, that is, ls can just always output the contents of the current working directory. You do not need to include "." or ".." in the list. You also do not need to mark directories with any special characters.
cd	No	No	Change to another directory relative to the current directory	Make sure you can accept the special directories "." (the current directory) and ".." (one directory up in the directory hierarchy). You do not need to support absolute paths. You do not need to support up-down paths (i.e., `cd../hello/world`). Important note: the cd command is the only command (other than exit) that does not need to participate in the piping mechanism, it can only output errors, it never accepts piped input or sends piped output. Hence, its implementation can be simpler than your implementation of the other commands.

Text Manipulation Commands				
cat	No	Yes	Output the entirety of one file to the output message queue	Unlike in UNIX, cat does not accept piped input. I.e. for your program, cat will always be first in a string of commands separated by pipes. As with cd, you do not need to support absolute paths or up-down paths (i.e., `cat ../hello/world/file.txt`).
grep	Yes	Yes	Read lines from piped input, and only output lines with a given search string	You do not need to do regular expression matching. You should do a simple case-sensitive string search, and count any line containing that search string as a match. You can assume that this argument (as well as all file names) will never contain >, or |.
wc	Yes	Yes	Read lines from piped input and output the number of lines, words and characters, separated with space	Note that this command heavily relies on figuring out when it is going to stop getting more input lines to count. A string with the line count, the word count, and character count for the input separated by spaces should be added to the output message queue. Words are considered strings separated by a space. Character count includes all characters in the input
uniq	Yes	Yes	Filters out duplicate lines from the input	Only permits lines of input to pass through if they have not been detected before (the Collections framework can be useful here).
head	Yes	Yes	Output the first 10 lines from the input to the output message queue.	If there are less than 10 lines of input, then all the input should be passed to the output queue.
tail	Yes	Yes	Output the last 10 lines from the input to the output message queue.	If there are less than 10 lines of input, then all the input should be passed to the output queue. The Java Collections framework has various data structure(s) that can be used to implement this efficiently.
>	Yes	No	Reads piped input, and writes it to a file specified after the > symbol	Make sure to utilize the java File class for this operation. If a file exists with the name given by the command, it should be overwritten by a new file created in its place.
General Commands				
exit	No	No	Quit the command line	Like cd, the exit command can be very simple, and it should just end your REPL and return from your main method after printing the "goodbye" message.

Error Reporting

Your program should be able to identify and report the errors shows below:

Invalid Property	Expected Behavior
Undefined Command	`> not-a-command` `The command [not-a-command] was not recognized.`

Missing Arguments	`> cat`
	`The command [cat] requires parameter(s).`
	`> cat file.txt | grep`
	`The command [grep] requires parameter(s).`

Invalid Piping Order	`> grep world | cat hello.txt`
	`The command [grep world] requires input.`
	`> wc | cat hello.txt`
	`The command [wc] requires input.`
	`> cd dir | wc`
	`The command [cd dir] cannot have an output.`
	`> cat file1.txt | cat file2.txt`
	`The command [cat file2.txt] cannot have an input.`

File/Directory Not Found	`> cat nonExistantFile.txt`
	`The file in the command [cat nonExistantFile.txt] was not found.`
	`> cd not-a-directory`
	`The directory specified by the command [cd not-a-directory] was not found.`

Note: Given the descriptions of commands and errors, you may assume (as you will notice when you run the tests) that your commands will not be provided too many parameters. E.g. a command that receives no parameters (like pwd) will not be provided any and that a command that receives one parameter (like grep) will not be provided any more than one. For this reason, you can parse a command with an argument as: [command] [space] [argument]. For instance, to parse cat file name with spaces.txt you can think of it having two components; the command cat and the argument file name with spaces.txt

Implementation Details

The project is intended to be developed and run on Eclipse (a Java IDE).

You will be provided the following files:

`SequentialREPL.java`:
Your REPL implementation should reside in this file, and should begin when the `main(String[])` method is called.

`Filter.java`:
An abstract class extended by the `SequentialFilter` class. **You should not modify this class.**

`SequentialFilter.java`:
An abstract class extending the `Filter` class. For each of the commands mentioned above, you need to extend this class. **You should not modify this class. A successful implementation of this project will demand a rigorous understanding of what this class is doing. Understand it before writing any code.**

`SequentialCommandBuilder.java`:
The `SequentialCommandBuilder` manages the parsing and execution of a command. It splits the raw input into separated subcommands, creates subcommand filters, and links them into a list. We have provided you with some suggested method declarations (a good way to break down the task), but you can feel free to approach this code as you see fit.

`AllSequentialTests.java, RedirectionTests.java, REPLTests.java, TextProcessingTests.java, WorkingDirectoryTests.java`:
These files contain multiple tests that examine your implementation of the various commands. **You should not modify these classes (besides for debugging purposes).** Note that a failure of any of these tests *will certainly lead to deducted points*. All perfect assignments will pass all tests but passing the tests does not guarantee a perfect score.

Implementation Hints

Your REPL *must* read user commands from `System.in`. There are two critical functions within this loop. (1) Correctly parsing commands in a way that allows the creation of piped filters. Using `Scanner` and `String.split` will help you with this task. (2) Identifying and reporting the different errors we mentioned above.

All commands that you have to implement must implement the abstract `SequentialFilter.java` class. The sequential filter is a convenient abstraction: it reads from an input queue (linked list), and writes to an output queue, by calling a method called `process()`. The queue that it writes to is the input queue for the next filter (if there is a next filter). The abstract class `SequentialFilter`, should greatly help you along in this project. `SequentialFilter` inherits from `Filter`, a class that is not displayed here, but is given along with this assignment.

Note that in implementing the filters, some of them can be simple classes of five or six lines of code who solely implement the `processLine(String)` method in the `SequentialFilter class`. (Examples of this include `grep`, >, and the default filter to

print to the command line, and `uniq`). Some other filters (`cat`, `ls`, `pwd`) will need to override the `process()` method of the `SequentialFilter class`, as they don't need to use the `processLine(String)` method to process any piped input.

The directory from which you start the command line program should be initialized as the current working directory that your Java program is initialized to. The working directory can be modified with the cd command. You use the command ls to list the contents of the current working directory. This will be a little more challenging than you might think because you are not allowed to modify the invoking environment's working directory. You'll need to manage a separate working directory for your shell in the static field `currentWorkingDirectory` in `SequentialREPL.java`. Note you should initialize this field inside `main()`.

Testing and Grading

Successful completion of this project is nearly impossible without appropriately testing against the provided JUnit Tests. If you cannot run the tests when you import your project, please see a TA at office hours as soon as possible.

Importing and exporting properly is crucial to allow us to grade your projects properly. Please import and export based on the instructions in the tutorial slides and on LATTE and submit the zip file that you exported from Eclipse. If you fail to export your project properly, you may be penalized.

You will need to include Javadocs together with the code you submit. The tutorial will also discuss Java Docs and how you can easily generate them. You should consult our guide on how to write and generate Javadocs on Latte and make sure you include the entire 'doc' folder in your submission. Failure to include Javadocs will be penalized.

There is a helpful Boolean variable called DEBUGGING_MODE in `AllSequentialTests.java`, which when set to true will not clean up after the created files and directories after unit tests are run. This can allow you to see the error in your output (files stored in the current working directory), and potentially understand any disconnect between our expectations and your code. Feel free to play around with the tests (modify them as is helpful for your debugging, etc.), **but know that you will be graded against the set of tests that we sent out originally.** Any attempt to tamper with the tests will be considered **a breach of academic integrity and** will result in point penalties and potentially further discipline.

Note on differences between Unit Tests and Integration Tests

Please note that though JUnit is being used to run this suite of tests, these are not unit tests, but integration tests. Unit tests test specific functions and small, incremental pieces of code. Our tests are predicated on the entirety of your code being complete, and this assumption breaks most of the conventions of unit test writing. To run the test suite, run `AllSequentialTests.java`.

Note on Platform Independency

Windows and Unix based operating systems differ on whether to use the "\" or "/" as their preferred file separator. In order to ensure your code will run in either case it is necessary to request this information at runtime, we have already included the necessary command in the starting code. `cs131.pa1.Filter` contains a static field `FILE_SEPARATOR` that will be set to this value for you to easily reference. Using a hardcoded "/" or "\" in your code instead of `Filter.FILE_SEPARATOR` to handle changing directories may result in a loss of points.

However, if your code is not platform independent when you submit it, do not fret; we will do our best to run your code on both a Windows and a Mac machine. This part of the assignment is not critical to the learning goal and will be worth less than five points of your final grade. If you want to make sure your code works on both machines, dual boot computers are available in the library to allow you to try your code in the other environment.

Note on Messages

In order for the test cases to properly evaluate your code, everyone must use the exact same messages for when your REPL loop starts, ends, reports an error, or requests a new command. We've included the expected messages in an enum for you to use to help with consistency. The enum `cs131.pa1.filter.Message` contains 10 messages that you will need to use throughout your project to report information to the user. Note that the 7 error messages contain a placeholder where the invalid command should be placed. Calling the function `with_parameter(String parameter)` for these 7 messages will replace the placeholder %s with parameter i.e. calling `Message.REQUIRES_INPUT("grep foobar")` will generate the String *"The command [grep foobar] requires input."* which is expected by the test cases. **Tests will fail if discrepancies exist in what you print to `System.out`, and a good first place to check a test failure is the exact strings being compared.**

Very Important Notes

- This is an individual assignment. You should only submit your own work. Any detected signs of collaboration and code copying will result in a 0 score for the assignment. Please check the academic integrity policies in the syllabus of the course.
- Late policy: Please check the syllabus for the late submission policies of the course.