CS 131: Operating Systems Programming Assignment 1, Part 1

Released: Tuesday, September 13th, 2016
Highly Encouraged Project Tutorial: Thursday, September 15th, 8PM
Due Friday, September 23rd, 11:55PM (on Latte)

Getting Started

Your assignment is going to be to build a basic command line environment for interacting with a file system. This command line environment (or shell) will be written in Java. The first part of this assignment is designed 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 do this while learning about Unix and practicing your Java-based System calls.

Common 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 to allow you to complete this project. After reading through these resources, 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	http://www.ee.surrey.ac.uk/Teaching/Unix/unixintro.html
Working Directory Commands	ls, cd, . , .. , pwd	http://www.ee.surrey.ac.uk/Teaching/Unix/unix1.html
Basic Text Operations	head, grep, wc	http://www.ee.surrey.ac.uk/Teaching/Unix/unix2.html
Redirection and Piping	Redirection, Piping, > |	http://www.ee.surrey.ac.uk/Teaching/Unix/unix3.html

REPL Loop

Your main program will include a loop that prints a simple prompt ("> "), accepts a command (or multiple commands separated by pipes), and prints the output from that command (if any). 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.
> head hello-world.txt
hello
world!
> head -1 hello-world.txt
```

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

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) Each function identifies and reports different errors as follows.

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

Invalid Arguments	`> head`
	`The command [head] is missing an argument.`
	`> head file.txt | grep`
	`The command [grep] is missing an argument.`

Invalid Piping Order	`> grep world | head hello.txt`
	`The command [grep world] requires input.`
	`> wc | head hello.txt`
	`The command [wc] requires input.`

File/Directory Not Found	`> head thisFileDoesNotExist.txt`
	`At least one of the files in the command [head thisFileDoesNotExist.txt] was not found.`
	`> cd not-a-directory`
	`The directory specified by the command [cd not-a-directory] was not found.`

Your REPL loop must support simple directory commands: ls, cd, and pwd. cd is the only command you will implement that does not use String queues, because it does not take piped input and does not produce output. You may handle this case without inheriting from the `SequentialFilter` class (see REPL classes defined below). ls and pwd do create piped output, even though they don't need piped input (the same is true for head).

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. This can be as simple as maintaining a String that keeps track of your present working directory.

Filter Commands

All commands that you have to implement within this half of the programming assignment 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). Shown below is the abstract class `SequentialFilter`, which 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.

On the next page of the assignment, there is a table detailing the commands you are responsible for implementing. A proper implementation of each should refer back to the correct error messages to print, and the behavior expected of each in a Unix-like system.

```
package cs131.pa1.filter.sequential;
import java.util.LinkedList;
import java.util.Queue;
import cs131.pa1.filter.Filter;

public abstract class SequentialFilter extends Filter {

     protected Queue<String> input;
     protected Queue<String> output;

     public void setPrevFilter(Filter prevFilter) {
          prevFilter.setNextFilter(this);
     }
     public void setNextFilter(Filter nextFilter) {
          if (nextFilter instanceof SequentialFilter){
               SequentialFilter sequentialNext =
               (SequentialFilter) nextFilter;
               this.next = sequentialNext;
               sequentialNext.prev = this;
               if (this.output == null){
                    this.output = new LinkedList<String>();
               }
               sequentialNext.input = this.output;
               } else {
                    throw new RuntimeException("Should not
```

```
                    attempt to link dissimilar filter types.");
               }
          }
     public void process(){
          while (!input.isEmpty()){
               String line = input.poll();
               String processedLine = processLine(line);
               if (processedLine != null){
                    output.add(processedLine);
               }
          }
     }
     public boolean isDone() { return input.size() == 0; }
     protected abstract String processLine(String line);
}
```

Note that in implementing the filters, some of them can be simple, five or six line classes who solely implement the `processLine(String)` method. (Examples of this include grep, > and the default filter to print to the command line). Some other filters (head, ls, pwd) will need to override the `process()` method, as they don't need to use the `processLine(String)` method to process any piped input. These classes and others may also need to override the `isDone()` method...

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. For this reason, your implementation of the cd command can be simpler than your implementation of the other commands.
Text Manipulation Commands				
head	No	Yes	Output the first 10 lines of one file	By default, it outputs the first 10 lines of the given file. It should accept an optional parameter specifying the number of lines to be displayed. (i.e. head -5 file.txt will output the first 5 lines of file.txt) Unlike the UNIX command head, you do not need to accept piped input to head. In other words, in your program, head will always be first in a string of commands separated by pipes.
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. Consider the possibility that there are no lines returned. Make sure that you utilize the `isDone()` method of this filter and the previous filter.
>	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.

Wrap It Up: What Do You Have And What You Need To Do

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. It should take its input from `System.in` (using a scanner is suggested).

`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. To implement 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 (Implication).

Testing and Grading

Successful completion of this project is nearly impossible without appropriately testing against the provided JUnit Tests. If you have difficulty adding the JUnit 3 or 4 library to the build path within the eclipse IDE, google that. If you still have difficulty, please see a TA at Office Hours as soon as possible.

To allow us to test without changing your code, please name the class which houses your main method `SequentialREPL.java`, this will be called when the test suite is run. Because we will be using unit tests that will compare your output (System.out) character for character against our samples, please make sure that your error messages are exactly the same as the ones given, and that there are no intentional or unintentional discrepancies (whitespaces, prompt, wording) between your result and the examples provided. **Tests will fail if these discrepancies exist, and a good first place to check a test failure is the exact strings being compared.**

There is a helpful Boolean variable called "DEBUGGING_MODE" (In the Test Suite), 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. Do not take this test design as an appropriate way to test the code that you are writing. Unit tests test specific functions and small, incremental pieces of code. These tests are predicated on the entirety of your code being complete, and this assumption breaks most of the conventions of unit test writing.

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 11 messages that you will need to use throughout your project to report information to the user. Note that the 8 error messages contain a placeholder where the invalid command should be placed. Calling the function `with_parameter(String parameter)` for these 8 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.

A Note on the Future

In this half of the assignment, you will be building a lot of code, but everything that is mentioned should be complete-able if you have taken COSI12 (focusing on inheritance and file processing), and COSI21A (Focusing on data structures and code design). Right now, all of this activity is Sequential (so in the command "head hello.txt | grep foo | wc", each filter is run in order, and terminates before the next one begins). In the next part of the assignment, you will use the skills that you will learn in this course to begin using concurrent stream processing, where each filter runs when it can, and the tasks of each filter are processed simultaneously (by multiple cores), or simply concurrently by one core.