CS380 Lab Assignment - Processes

This assignment consists of creating a shell interface in Java by using processes. Some details and code to help with the assignment are provided next. Following this details on what you need to do in this assignment are provided.

Before you begin the assignment:

Recall that a shell interface provides the user with a prompt, after which the user enters the next command. The example below illustrates the prompt 'prompt>' and the user's next command: 'cat Prog.java'. This command displays the file 'Prog.java' on the Terminal using the 'cat' command.

```
prompt> cat Prog.java
```

Perhaps the easiest technique for implementing a shell interface is to have the program first read what the user enters on the command line (here, cat Prog.java) and then have the program create a separate external process that performs the command. You can create the separate process using the code provided below:

```
import java.io.*;

public class AProcess
{
	public static void main(String[] args) throws IOException {
		if (args.length != 1) {
			System.err.println("Usage: java OSProcess <command>");
			System.exit(0);
		}

		// args[0] is the command
		ProcessBuilder pb = new ProcessBuilder(args[0]);
		Process process = pb.start();

		// obtain the input and output streams
		InputStream is = process.getInputStream();
		InputStreamReader isr = new InputStreamReader(is);
		BufferedReader br = new BufferedReader(isr);

		String line;
		while ( (line = br.readLine()) != null)
			System.out.println(line);

		br.close();
	}
}
```

The following code contains the basic operations of a command-line shell. The main() method presents the prompt '*prompt>*' and waits to read input from the user. The program is terminated when the user enters <Control><C>.

```
import java.io.*;

public class ABasicShell
{
	public static void main(String[] args) throws java.io.IOException {
		String commandLine;
		BufferedReader console = new BufferedReader(new
InputStreamReader(System.in));

		// we break out with <control><C>
		while (true) {
			// read what they entered
			System.out.print("prompt>");
			commandLine = console.readLine();

			// if they entered a return, just loop again
			if (commandLine.equals(""))
				continue;

			/**
			  The basic steps are:
			  (1) parse the input to obtain the command
			      and any parameters
			  (2) create a ProcessBuilder object
			  (3) start the process
			  (4) obtain the output stream
			  (5) output the contents returned by the command
			 */
		}
	}
}
```

Today's Assignment:

This assignment has 3 parts to it (these parts are described in more detail below):
(1) creating the external process and executing the command in that process,
(2) modifying the shell to allow changing directories,
(3) adding a history feature.

Part 1: Creating an External Process
The first part of this assignment is to modify the main() method for the above provided Shell code so that an external process is created and executes the command specified by the user. Initially, the

command must be parsed into separate parameters and passed to the constructor for the ProcessBuilder object. For example, if the user enters the command

prompt> cat Prog.java

the parameters are (1) cat and (2) Prog.java, and these parameters must be passed to the ProcessBuilder constructor. Perhaps the easiest strategy for doing this is to use the constructor with the following signature:

public ProcessBuilder (List<String> command)

A java.util.ArrayList — which implements the java.util.List interface — can be used in this instance, where the first element of the list is cat and the second element is Prog.java. This is an especially useful strategy because the number of arguments passed to Terminal commands may vary (the cat command accepts one argument, the cp command accepts two, and so forth).

If the user enters an invalid command, the start() method in the ProcessBuilder class throws an java.io.IOException. If this occurs, your program should output an appropriate error message and resume waiting for further commands from the user.

Part 2: Changing Directories

The next task is to modify your program so that it changes directories. In Mac systems, we encounter the concept of the *current working directory,* which is simply the directory you are currently in. The cd command allows a user to change current directories. Your shell interface must support this command. For example, if the current directory is /usr/tom and the user enters cd music, the current directory becomes /usr/tom/music. Subsequent commands relate to this current directory. For example, entering ls will output all the files in /usr/tom/music. The ProcessBuilder class provides the following method for changing the working directory:

public ProcessBuilder directory(File directory)

When the start() method of a subsequent process is invoked, the new process will use this as the current working directory. For example, if one process with a current working directory of /usr/tom invokes the command cd music, subsequent processes must set their working directories to /usr/tom/music before beginning execution. It is important to note that your program must first make sure the new path being specified is a valid directory. If not, your program should output an appropriate error message.

If the user enters the command cd, change the current working directory to the user's home directory. The home directory for the current user can be obtained by invoking the static getProperty() method in the System class as follows:

System.getProperty("user.dir");

Part 3: Adding a History Feature

Many shells provide a *history* feature that allows users to see the history of commands they have entered and to rerun a command from that history.

The history includes all commands that have been entered by the user since the shell was invoked. For example, if the user entered the history command and saw as output:

0 pwd
1 ls -l
2 cat Prog.java

the history would list pwd as the first command entered, ls -l as the second command, and so on.

Modify your shell program so that commands are entered into a history. (Hint: The *java.util.ArrayList* provides a useful data structure for storing these commands.)

Your program must allow users to rerun commands from their history by supporting the following three techniques:

1) When the user enters the command history, you will print out the contents of the history of commands that have been entered into the shell, along with the command numbers.
2) When the user enters *!!*, run the previous command in the history. If there is no previous command, output an appropriate error message.
3) When the user enters *!<integer value i>,* run the *i*th command in the history. For example, entering *!4* would run the fourth command in the command history. Make sure you perform proper error checking to ensure that the integer value is a valid number in the command history.

<u>Assignment Submission Instructions:</u>

Submit your completed assignment through Moodle. You should submit (1) a zip file containing your Java code, and (2) a zip file containing screenshots showing your code running (the output from running your code through e.g. Eclipse).

Before submitting your code ensure that it is well commented and formatted (marks will be deducted for code that is not well commented and formatted).

Assignment submission deadline is available on Moodle. Penalties will be imposed on late submissions.