Preview:

Working on both Problems 2 and 3 should increase your understanding of pipelines. In Problem 2, you are given a line-scrambled program that would demonstrate pipes if it were unscrambled. The pipeline is formed by each new process forking its child to be the next process in the pipeline. You must unscramble the program. In Problem 3, you are asked to think about how to implement the *fanout forking* for creating a pipeline of commands.

Problem 1 (0 Points)
Familiarize yourself with the sections of The Gnu C Library Reference Manual that pertain to implementing the *rsh* shell.

Problem 2 (6 Points)
The course Web page contains the file `npipe3-puzzle.c` and its header file `stdinc.h`. At one time, the program solved the `npipe` problem described below but for the limited case of \( N = 3 \) processes (i.e., a pipeline of two pipes). But the student dropped the `npipe3.c` source code on the floor, and some lines got scrambled.

Here is the description of the more general `npipe` program. It is called like this:

```
npipe [-n N]
```

where \( N \) indicates the number of instances of `npipe`. Basically, when run it forms a pipeline with \( N - 1 \) pipes connecting \( N \) instances of the `npipe` process: ”`npipe | npipe | ... | npipe`” The \( k \)th process creates the pipe to the \( (k + 1) \)th process and forks the \( (k + 1) \)th process. As a verification that each process outputs the same bytes, each process computes the sum of all bytes treating each byte as an unsigned integer value. This *checksum* is printed on `stderr` by the main routine before it exits.

The program `npipe3` is just `npipe` with \( N = 3 \). You are to unscramble the code so that it will compile and run properly. We know the following:

- See the comments in the beginning of the `npipe3-puzzle.c` code for the usage syntax.
- In its correct order, the program compiled using ”`g++ -g -o npipe3 npipe3.c`” and ran correctly on a Linux 2.6 system. There may be minor changes that are needed to get it to compile on other systems.
- The file `stdinc.h` is correct.
- The only lines that got scrambled are in the following functions: `pipe_stdout`, `pipe_stdin`, `do_plumbing`, and `do_jo`.
- Debugging output can be enabled by uncommenting the line of code `debug = 1;` in the main routine.
You may have to do some or all of the following:

- Review the syntax and semantics of `pipe()` and `dup2()` system calls and how they are suppose to be used.
- Read the code (and comments). Note that none of the comments were reordered when the source code got scrambled.
- Insert code to output additional information.
- Run a debugger against the code.
- Do a simple test with only one process and no pipe.
- Do a simple test with only two processes and one pipe.

Submit the following:

- The source program listing for the working version
- The output for some test cases when in debug mode
- An explanation of why the output indicates that the program is running correctly
- The output of the command "`diff npipe3-puzzle.c npipe3.c`" where `npipe3.c` is the working code and `npipe3-puzzle.c` is the original code.

**Problem 3 (8 Points)**

Consider the following three `xssh` pipelines:

```
ls -l /usr/bin | grep Dec | sort -n +4
ls -l /usr/bin | grep Dec | sort -n +4 > xxx
ls -l /usr/bin | grep Dec | sort -n +4 > xxx &
```

a) Give the pseudo-code required to evaluate the first pipeline in which `xssh` creates all pipelines and forks all processes in the pipeline. Number each line of the pseudo-code (the line numbers will be used in Part b). Assume that only non-built-in commands are in the pipeline.

b) A correct solution should exhibit each of the properties listed below. For each property, explain which line(s) of your pseudo-code will lead to the property listed.

1) `xssh` creates one child process for each command in the pipeline.
2) `xssh` creates one less pipeline than the number of commands.
3) Child \( k \) redirects `stdout` to `pipe \( k \)` for \( k = 0, 1 \) if counting from 0.
4) Child \( k \) redirects `stdin` to `pipe \( k - 1 \)` for \( k = 1, 2 \) if counting from 0.
5) `xssh` waits for each child process it creates.

c) Explain how the pseudo-code in Part a would need to be modified to support the `stdout` redirection shown in the second pipeline.

d) Explain how a shell interpreter should handle a pipeline command that needs to be run in the background. Begin by listing the difficulties involved and the key ideas you will apply towards addressing these difficulties.