CS 111: Operating System Principles
Lab 1

Pipe Up 3.0.0

Rustem Can Aygun, Yadi Cao, Victor Zhang
Derivative document by: Jonathan Eyolfson
April 14, 2023
Due: April 28, 2023 @ 11:55 PM PT

In this lab you'll be writing the low level code performed by the pipe (|) operator in shells. Users will pass in executable names as command line arguments and you will execute each one in a new process (again similar to a shell). You're expected to understand your own implementation and test it yourself.

Additional APIs. Given you are just using the executable names, you may use the C library helper functions for `execve`, such as `execlp`. `execlp` will let you skip using string arrays (using C varargs), and it will also search for executables using the `PATH` environment variable. You may notice your program hanging waiting for input. You must call `close` on any file descriptors not explictly used in your process. Failure to call `close` will inform the operating system you are not done with it, and it will never return end-of-file from a `read` system call.

Starting the lab. Download the lab1 skeleton code tar file from the course Bruinlearn page. You should be able to run `make` in the `lab1` directory to create a `pipe` executable, and then `make clean` to remove all binary files.

Files to modify. You should only be modifying `pipe.c` and `README.md` in the `lab1` directory.

Your task. You should execute the programs in `argv[1]`, ..., `argv[argc - 1]` as new processes. You also need to create a pipe between two subsequent processes. For example, a pipe should connect `argv[1]`'s standard output to `argv[2]`'s standard input (if there are at least two procsses). The standard input of first new process must be the same as the standard input of the parent process (`pipe`). Also, the standard output of your last new process must be the same as the standard output of the parent process. You should be able to handle between 1 to at least 8 programs (more is okay). All standard errors should be the same as the parent's standard error. You do not need to handle passing additional command line arguments to every individual new process. All your processes should be a direct child of the original process that starts executing `main`. Finally, fill in your `README.md` so that you could use your program without having to use this document.

Errors. Your program should (of course) handle errors from all function calls you make. Your program should exit with the proper `errno` of the failing call. It is okay to exit as soon as you find an error, without any error recovery. If there are no programs as command line arguments, your program should exit with errno `EINVAL` (invalid argument). Your program should work with a single program as a command line argument. Your program should not create any orphan processes (you must `wait`).

Tips. You should come up with smaller subtasks yourself. One approach may be to execute one program from the command line, then multiple programs independently. Afterwards set up your pipe between two processes, then multiple processes. Start small then work big.

Example output. Your output should be the same as if you were to use | between each program in your shell.

```
> ./pipe ls
Makefile  pipe  pipe.c  pipe.o  README.md
> ./pipe ls cat wc
      5       5      38
```

The last command should have the same output of: `ls | cat | wc`.

Testing. There are a set of basic test cases given to you. We'll withhold more advanced tests which we'll use for grading. Part of programming is coming up with tests yourself. To run the provided test cases please run the following command in your lab directory:

```
python -m unittest
```

Grading. The breakdown is as follows:

- 90% code implementation
- 7% documentation in README.md
- 3% lab1 discussion notes

Submission.

- All lab submissions will take place on BruinLearn. You will find submission links for all labs under the Assignments page.
- Your submission should be a single tarball that includes only the following files: pipe.c, README.md and discussion_notes.txt(include this one if you attended to the discussion). The tarball should be named YOURUID-lab1-submission.tar, where YOURUID is your 9 digit UID without any separators. You can create this tarball by running the command make tar inside your lab1 directory (note: running make tar will overwrite the previously created tarball). Any submission that does not follow the submission guideline will receive -15 pts. (Note: if you make multiple submissions, Bruinlearn will automatically append the filename with -SOMENUMBER, you don't need to worry about this.)
- Your submission will be graded solely based on your last submission to BruinLearn, without any exceptions. **Please double check your submission to make sure you submit the correct version of your implementation.**