Introduction to Operating Systems

Homework 2: Pipe Flow

Due 10/20/2024 @ 11:59pm

Introduction

In this homework you will chain processes together in a way similar to how linux shell does. You will be doing this in your own environment – not xv6. You can use Linux, WSL (Windows Subsystem for Linux) or Mac. This is hopefully great news for you since you can use all the standard libraries. You also can work with a partner (like for HW1).

Let's start with a few examples (these are run on a Mac under `bash` shell):

Unset
```
kamenyotov@Kamens-Air ~ % ls | wc
      25      37     418
```

In this example we run the `ls` command and redirect its standard output to feed it as a standard input to the `wc` command. As a result `wc` computes that I have 25 entries in the current directory which when printed contain 37 words and 418 characters.

Unset
```
kamenyotov@Kamens-Air ~ % ls > result.txt
kamenyotov@Kamens-Air ~ % ls -la result.txt
-rw-r--r--@ 1 kamenyotov  staff  429 Sep 28 20:03 result.txt
```

In this example we redirected the standard output of `ls` into the file `results.txt` instead. You can open the file and examine it!

```
Unset
kamenyotov@Kamens-Air ~ % wc < result.txt
      26      38     429
```

In this example we redirected the standard input of wc to come from the `result.txt` file above.

The three examples above are by far the most common input and output redirections that people use on unix. There are some more elaborate ones that are more rarely used but still useful.

```
Unset
kamenyotov@Kamens-Air ~ % mkdir a
kamenyotov@Kamens-Air ~ % mkdir a
mkdir: a: File exists
kamenyotov@Kamens-Air ~ % mkdir a | wc
mkdir: a: File exists
       0       0       0
kamenyotov@Kamens-Air ~ % mkdir a |& wc
       1       4      22
```

In this example we first create a directory `a`. Then we try to create it again, and we get an error (since the directory already exists). That error is printed on standard error, not standard out. You can see that when we try to redirect the standard output to wc, the error is still printed and wc says it has read 0 lines. Finally when we use `|&` to redirect instead of simple `|` we see that the error is also redirected.

When `|&` is used for redirection, the standard error is combined with the standard output before it is being redirected. A different way to do this would be this:

```
Unset
kamenyotov@Kamens-Air ~ % mkdir a 2>&1 | wc
       1       4      22
```

Here we redirect file descriptor 2 (standard error) to file descriptor 1 (standard output) and then redirect standard output to wc.

Here is an even more complicated example:

```
Unset
kamenyotov@Kamens-Air ~ % echo "f o o" > foo.txt
kamenyotov@Kamens-Air ~ % cat foo.txt
f o o
kamenyotov@Kamens-Air ~ % cat foo.txt | sed 's/o/u/g'
f u u
kamenyotov@Kamens-Air ~ % ( cat foo.txt ; cat foo.txt | sed 's/o/u/g' )
f o o
f u u
kamenyotov@Kamens-Air ~ % ( cat foo.txt ; cat foo.txt | sed 's/o/u/g' ) | wc
       2       6      12
```

First we create a file `foo.txt` which contains `f o o`. Then we “massage” this content by replacing the `o`s with `u`s. This is achieved by the `sed` command and a handy regular expression.

We then concatenate the output of `cat foo.txt` and `cat foo.txt | sed ‘s/o/u/g’`. This is achieved by the `( <command_1> ; <command_2> )` syntax.

Redirections can get complicated and the syntax is pretty cryptic… as you can see. If you are interested you can read more at https://www.gnu.org/software/bash/manual/html_node/Redirections.html.

There is also the concept of process substitution (https://tldp.org/LDP/abs/html/process-sub.html) which makes things even more hairy.

Task

In this homework you will create a more user-friendly way to chain processes than bash redirections!

We will make a language of our own which we can use to define commands and chain them anyway we line in complicated data-flow graphs!

Here is how we can do the first example above `ls | wc` in our new language. This is the content of the `filecount.flow`:

```
Unset
node=list_files
command=ls

node=word_count
command=wc

pipe=doit
from=list_files
to=word_count
```

We will implement a program `flow` to run these. The above can be ran with:

```
Unset
./flow filecount.flow doit
```

The first parameter is the description file and the second parameter is the action from it to run.

Here is the example program for the complicated example we had in the end, i.e. `( cat foo.txt ; cat foo.txt | sed 's/o/u/g' ) | wc` which we will call `complicated.flow`:

```
Unset
node=cat_foo
command=cat foo.txt

node=sed_o_u
command=sed 's/o/u/g'

pipe=foo_to_fuu
from=cat_foo
to=sed_o_u
```

```
concatenate=foo_then_fuu
parts=2
part_0=cat_foo
part_1=foo_to_fuu

node=word_count
command=wc

pipe=shenanigan
from=foo_then_fuu
to=word_count
```

And we can run it with `./flow complicated.flow shenanigan`.

Basically you are writing an interpreter for these flow graphs that reads the graph description from a file and then creates the appropriate processes and stitches them together the way a shell would do.

The parts that you need to implement are as follows:

- `node`
- `concatenate` – this is for running a sequence of nodes and producing their outputs sequentially

The above **CONCEPTS IMPLEMENTED** is what you need for full credit. What I list below is for extra credit.

Extra Credits

- `stderr` – this is to extract the standard error output of a node, passed to the `from` attribute
- `file` – this is for accepting input or output from a file, filename passed to the `name` attribute
- … more to come …

Extra Credit Examples

1. Error Handling Example:

```
node=mkdir_attempt
command=mkdir a

node=word_count
command=wc

stderr=stdout_to_stderr_for_mkdir
from=mkdir_attempt

pipe=catch_errors
from=stdout_to_stderr_for_mkdir
to=word_count
```

Explanation: mkdir_attempt tries to create a directory a. If it fails, the error message (from stderr) is sent to stdout_to_stderr_for_mkdir, which uses wc to count lines, words, and characters in the error.

2. File Handling Example (Input and Output):

```
node=read_file
command=cat

file=input_file
name=result.txt

node=word_count
command=wc
```

```
pipe=read_pipe
from=input_file
to=read_file

pipe=process_pipe
from=read_pipe
to=word_count
```

Explanation: `cat` reads from `result.txt` and pipes its contents to `wc`, which counts lines, words, and characters.

2024-10-14 Clarifications (after Kamen Yotov chatted with TAs)

- Do you need to implement all redirections?
    - All redirections, at the end of the day, can be handled by pipes.
    - Let's say you wanted to do something like `ls > foo.txt`
    - There is a command in linux `tee` which reads `stdin` and writes `stdout` *AND* also writes to a file.
    - You can do `ls | tee foo.txt` and the effect of that would be that the result of `ls` will be printed on the screen *BUT ALSO* written to `foo.txt`.
    - You can do this with the following flow file:

```Unset
node=list_dir
command=ls

node=tee_to_foo
command=tee foo.txt

pipe=ls_to_foo
from=list_dir
to=tee_to_foo
```

- OK, but what about input redirection?
- Well, `cat foo.txt` will read a file and send the content to `stdout`
- So if you wanted to do `wc < foo.txt` you can do the equivalent `cat foo.txt | wc`.
- How do you translate this to `.flow` is left as an exercise to the reader 🙂
- OK, but what about standard error (i.e. `2>&1` and the like…)
  - Well, it is possible but that is part of the extra credit, so figure it out.