CSE3100 - Introduction to System Programming
Assignment #5

The purpose of this assignment is to experiment with process management APIs and pipe APIs. The best way to do this is to build the part of a "baby" shell responsible for executing sub-commands and creating pipelines. Remember that a basic command can use file redirections while a pipeline chains a sequence of commands, each one susceptible of carrying some arguments. The bulk of the shell code is distributed as part of the handout. All you have to focus on is the part responsible for implementing the proper behavior. This does not represent a lot of code, but it can be challenging to understand the subtleties.

Exercise 1. ***Basic command (40 points)*** *Consider a simple shell command relying on an external executable. For instance, the* `cat` *command (found as the executable* `/bin/cat`*) shown during the lab is used to "copy" its standard input on its standard output. If given an argument, it will instead copy the file whose name is given in argument on its standard output. For instance, the command*

```
cat hello.txt
```

puts on the standard output the content of the file named `hello.txt`*. Similarly, the fragment*

```
cat hello.txt > foo.txt
```

uses a file redirection to send the standard output of `cat` *to the file named* `foo.txt`*, effectively creating a copy of* `hello.txt` *in* `foo.txt`*. Your first task is to find the function on line 337 of* `shellp.c`

```
int basicExecute(char* com,int mode,char* input,char* output,char** args)
{
   //TODO: Implement this basic command.
   return 1;
}
```

and implement its body. The interface of the function is quite straightforward and the arguments are described below

com *the full name to the binary of the executable to run*

mode *an integer whose bits indicate the kind of file redirections. When equal to* `R_NONE`*, there are no redirections. If bit* `R_INP` *is set, the input is redirected. If bit* `R_OUTP` *is set the output is redirected. If bit* `R_APPD` *is send, the output is redirected and we should append at the end of the specified file.*

input *an argument which is only defined if* `R_INP` *is set.*

output *an argument which is only defined if* `R_OUTP` *or* `R_APPD` *is set.*

args *an array of strings holding the arguments that must be passed to the command* ***com****. The first argument is the executable path and the last one is NULL.*

The first step is to simply setup the logic for forking and executing the child process. The second step is to use the `mode` *argument along with* `input` *and* `output` *to setup the file redirections (if present).*

Once you've implemented the `basicExecute` *function, you should be able to execute single commands:*

```
$ ./shell
% echo hello > thing.txt
>  [thing.txt]
CORE: echo
        args[0] = echo
        args[1] = hello
```

```
        args[2] = (null)
Stages:
% cat thing.txt
CORE: cat
        args[0] = cat
        args[1] = thing.txt
        args[2] = (null)
Stages:
hello
```

Exercise 2. ***Pipeline command (60 points)***

A pipeline is simply a sequence of external programs chained together to form a single construction. The chaining occurs when the standard output of stage i of the pipeline is fed to the standard input of stage $i+1$. Consider that each stage of the pipeline consist of an external command (as in question 1) that can take a collection of arguments. Its input is coming from a pipe fed by the previous pipeline stage and its output feeds the next stage of the pipeline.

Your specific task is to implement the function `setupCommandPipeline` *found on line 348 of* `shellp.c`.

```
int setupCommandPipeline(Command* c)
{
   // TODO: Implement the pipeline commmand.
   return 1;
}
```

The command object is an abstract data type defined in the header file and reproduced in Figure 1. As you can observe, the Figure defines two structures. The `Command` *structure specifies the entire pipeline. Note that the arguments to each stage are embedded in the* `_stages` *array and the array in the* `Command` *are not used for pipelines.*

Consider for instance the pipeline

```
$ cat whitman.txt|tr -s "[:space:]" "\n"|sort|uniq|wc > howMany.txt
```

shown during the lab. It contains 5 stages.

1. *The first stage puts on the standard output of the stage the content of the file* `pairs.txt`.
2. *The second stage takes that as input and replaces every space by a single line-feed and sends the result on its standard ouptut*
3. *The third stage sorts all those lines in alphabetical order*
4. *The fourth stage eliminates duplicate entries and puts the "cleaned" version on its standard output*
5. *The fifth and last stage counts the number of lines in that pipeline and sends that final output of the entire pipeline into a file named* `howMany.txt`.

Now how some stage have no arguments while stage 2 has three arguments. Also note how the entire pipeline output is redirected to a file. The `Command` *instance passed to* `setupCommandPipeline` *contains an exact description of the pipeline. Note how redirections are global to the pipeline and not local to a stage. Your code must setup the pipes connecting all the processes that implement the pipeline. The parent process (the shell) must orchestrate the creation of all the processes and their coordination through the pipes as well as possibly setting up the redirections. It may be necessary to add attributes (fields) to the* `Command` *or the* `Stage` *ADT. Feel free to do so.*

Enjoy!

```
enum Kind {
  noCMD,exitCMD,cdCMD,pwdCMD,linkCMD,rmCMD,basicCMD,pipelineCMD
};

typedef struct Stage {
   int      _nba;  // number of arguments for this stage
   char**  _args;  // array of arguments for this stage
   // Feel free to add fields as needed
} Stage;

typedef struct Command {
   char*     _com;  // actual command line
   enum Kind _kind; // an enum value telling which kind of command this is
   int      _mode;  // bitmask indicating the redirections
   char*   _input;  // input file redirection for the whole commmand / pipeline
   char*  _output;  // output file redirection for the whole commmand / pipeline
   char**   _args;  // array of arguments (when a simple command, ignore for pipeline)
   int       _nba;  // number of arguments (when a simple command, ignore for pipeline)
   Stage** _stages; // Array of stages (only for pipelines)
   int       _nbs;  // numbers of stages in _stages (only for pipelines)
   // Feel free to add fields as needed
} Command;

// A pipeline "object" contains
// 1. An array of nbStages stages
// 2. A mode flag to determine whether
//           - the pipeline input is redirected
//           - the pipeline output is redirected
// Each stage is a structure that encapsulates
// - The number of arguments
// - Each argument (in string form)
// BY DEFAULT THE MAIN PROGRAM CALLS THE PRINT METHOD OF THE COMMAND
// OBJECT. So you can see how the pipeline description is represented/stored
// Your objective:
// * Implement the execute method to create and execute the entire pipeline.
// Hint: all systems calls seen in class will be useful (fork/exec/open/close/pipe/...)
// Good Luck!
```

Figure 1: The Command ADT