CS342 - Spring 2019
Project #1
A Simple Shell Program

Assigned: **Feb 15, 2019.**
Due date: **March 2, 2019, 23:55.**

Objectives

- Practice process creation and execution in Linux.
- Practice interprocess communication (IPC), pipes.
- Practice designing and doing experiments, statistics knowledge.

This project will be done individually. You will use C and Linux.

Project Description

In this project you will develop a simple shell program, a command line interpreter, called `bilshell`. It will work in two modes: interactive mode and batch mode. In *interactive mode* your shell will provide a prompt string, like `bilshell-$:`, to the user where the user will type a command name, i.e., a program name, with zero or more parameter strings and your shell will execute the command. In *batch* mode your shell will read the commands from a file and will execute them one after another. Your shell will be invoked as follows:

bilshell N $inputfilename$

The $inputfilename$ parameter is the name of an ascii text file containing commands. There is one command per line. The $inputfilename$ parameter is optional. If omitted, the shell will run in interactive mode; if specified it will run in batch mode. The parameter N is the number of characters to receive in each read operation and will be used for compound command execution that will be explained later. An example invocation of the program can be:

`bilshell 1 infile.txt`

A command to execute will include a command name, i.e., a program name, and zero or more parameters. An example command can be "cp file1.txt file2.txt".

When such a command is entered by a user or read as a line from an input file, your shell will divide it into arguments, i.e., strings, where argument 0 is the command name, argument 1, if any of course, is the first parameter to the command, argument 2 is the second parameter to the command, and so on.

When started, your shell will run as a process, i.e., main parent process, and will wait for an input command line. When user gives an input line, the line will be separated into arguments and a child process will be created to execute the command.

For this the main process will use the `fork()` system call. In the child process, exec() system call will be used to finally execute the program. There are various exec related library functions. You can use `execv()` or `execvp()` for example. It takes the pathname of a program to execute and an array of argument strings. Please read the man page of exec system call. After creating the child process, the parent process will wait. When command is executed and child process terminates, the parent process will return from waiting and will provide another prompt string to the user so that the user can type another command line; or if commands are taken from an input file, the parent will read another command line from the input file and will execute that command again in another child process.

Your shell will also support composition of two commands where the output of one command will be given to another command as input. For example, there is “ps aux” program in Linux that is listing the current processes, and there is “sort” program in Linux that is sorting a text file. When we write “ps aux | sort” in Linux shell, it prints to the screen the sorted list of processes. Similarly, when we would write such a command line in your shell, it should also print a listing of processes in sorted order. Such a command line consists of two commands, with possible parameters, separated with | symbol. The symbol | is called the pipe symbol. Your shell will support use of only *one* pipe symbol in a command line, hence the compound execution of *two* commands in a command line.

When a command line with pipe symbol is entered in your shell or read from the input file, the main shell process with create two child processes. Two fork calls are needed to do this. In each child we need to use the exec system call to execute the respective program. The output of one program, that would normally go to screen, i.e., to standard output, will now go as input to the second program, which would normally receive the input from a user or from a file. This requires communication (IPC) between these processes.

Your shell will provide communication between two child processes executing two programs by use of Linux unnamed *pipes*. But unlike a normal Linux shell program that creates a single pipe between two child processes directly so that the output of one child process is fed directly as input to the other, your shell will use *two* pipes. For this, your main process will create two pipes that can be called as pipe1 and pipe2, for example. The output of the first child process will be directed to the first pipe, from where your main process will read the incoming stream of characters. Your main process will write those characters to the second pipe from where the second child process will take the input. This is illustrated in Figure 1. Your main process will read from pipe1 and write to pipe2 N characters at a time using the `read` and `write` system calls. N is the value that is given as an argument to your shell program when it is started. N can be between 1 and 4096.

Main process will create the pipes using the `pipe()` system call. Main process will then create the child processes using the fork system call. Then exec system call will be called in each child to run the respective program. Before calling exec, in each child, I/O redirection will be done to direct the output or input. For the first child, the output will not go to standard output anymore but will be redirected to pipe1. For the second child, the input will not come from standard input anymore but will be directed from the second pipe. This can be done by use of the `dup2()` system call.

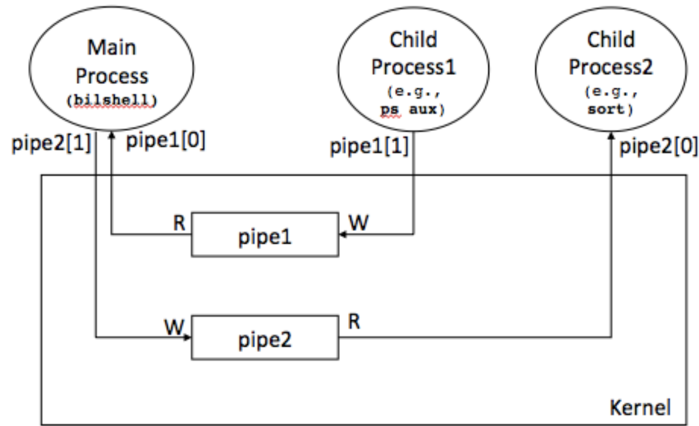

Figure 1: Compound command execution and communication of child processes through the parent.

It is used to duplicate a file descriptor. For the first child, the write end of the pipe1 will be duplicated to file descriptor 1. In this way whenever child 1 program would access file descriptor 1, i.e., standard out descriptor, it would access the write end of pipe1. Integer 1 is always the file descriptor corresponding to standard output. This can be done by the statement `dup2 (pipe1[1], 1)` (see Figure 2, showing the open file tables of parent and child processes before and after dup2 call). For the second child, the read end of pipe2 will be duplicated to file descriptor 0. Integer 0 is always the file descriptor corresponding to standard input. This can be done by the statement `dup2 (pipe2[0], 0)`.

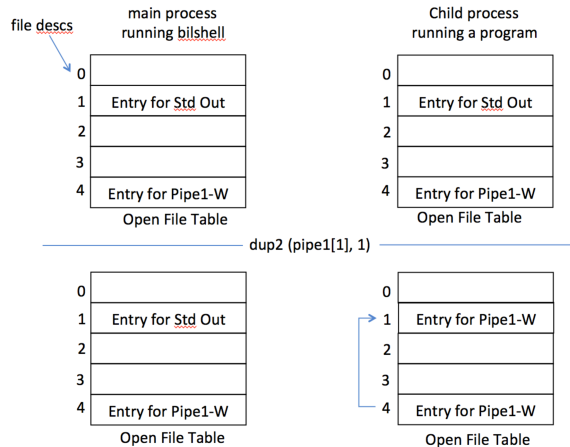

Figure 2: Effect of dup2 call.

After creating child processes, the main process will read from pipe1 and will write to pipe2. Do not forget to close the unused ends of the pipes at the main process. When a child terminates, the ends of the pipes that are used by the child are closed automatically as well.

For compound command execution, since the main process is on the data flow path from one chid process to the other, it can keep some statics about the trans-

ferred data. Count the number of bytes transferred through the pipes for compound commands. Count also the number of read/write operations performed from and to pipes. Print the count to screen after compound command execution has finished. The output format should be as in the following example:

```
character-count: 15000
read-call-count: 15000
```

Experiments

Now do the following simple experiment. Assume we are wondering about the effect of N on the performance of a compound command execution. Write two simple programs "producer M" and "consumer M" as commands to be compounded. When separately executed, the producer will just print M random alphanumeric characters to screen one by one or N characters at a time. You can do this again by using the write system call where file descriptor is standard out, i.e., 1. And consumer will just read M characters from standard input one by one or N characters at time. Now run these programs in a compound fashion in your shell and measure the time it takes. That means execute the following command, for example, from an input file in your shell: "producer 1000000 | consumer 1000000". Using the `time` command, measure the running time of your shell executing just this compound command from a file. Repeat this for various values of N and for a big enough M. Report the results in a `report.pdf` file and try to comment on them.

Submission

Submit a pdf file as your report discussing your experiments. Your report will include the results, your interpretations and conclusions, and also the code of the simple programs that your have written. Put your report.pdf file and all other files (bilshell.c and a Makefile and a README.txt file) into a directory named with your ID. Then tar and gzip the directory. For example a student with ID 21404312 will create a directory named 21404312 and will put the files there. Then he will tar the directory (package the directory) as follows:

```
tar cvf 21404312.tar 21404312
```

Then he will gzip the tar file as follows:

```
gzip 21404312.tar
```

In this way he will obtain a file called 21404312.tar.gz. Then he will upload this file in Moodle.

Late submission will not be accepted (no exception). A late submission will get 0 automatically (you will not be able to argue it). Make sure you make a submission one day before the deadline. You can then overwrite it.

Tips and Clarification

- Will be posted to course website and can be sent via piazza as well.