CS 350 Operating Systems, Spring 2024
Programming Project 3 (PROJ3)

Out: 10/23/2024, WED
Due date: 11/10/2024, SUN 23:59:59

In this project, you will implement your own shell terminal based on a command-line parser program in xv6. The goal of this project is to practice multi-process programming and better understand the OS process API.

This is a group project, and is expected to be completed with your group members. You can choose to divide and conquer the assignment, or work as a group to solve problems together. Regardless, each group member is responsible for submitting their own code on Brightspace.

This project contributes 6.25% toward your final grading.

1 Baseline source code

For this and all the later projects, you will be working on the baseline code that needs to be downloaded from Brightspace. This code should be familiar to you from various labs where you implemented features using fork(), exec(), wait(), and pipe(). Please extract the zip file provided on Brightspace and that code (not your project or lab code).

2 The baseline code - a command-line parser program

This section introduces the command-line parser program (named "sh"), which comes standard with xv6. This parser program supports the following functionalities.

- Basic command parsing.
- Parsing of multiples of commands in the same line.
- Parsing of command a pipeline.
- Parsing of command input and output redirection.
- Parsing of command background execution.

2.1 Basic command parsing

A command is a sequence of words separated by blanks, terminated by a *control operator*. The first word generally specifies a command, which can be a shell command program or a user program, to be executed, with the rest of the words being that command's arguments. For example, in the following command

```
echo hello operating systems
```

`echo` is a shell command program and "hello", "operating", and "systems" are the three arguments to the program. The following control operators should be supported by the command-line parser.

Control operator	Meaning
<newline>	Command delimiter
`;`	Command delimiter
`|`	Pipe
`&`	Background execution

2.2 Parsing of multiple commands in the same line

Multiple separate commands/pipelines can be put in the same line when delimited by the control operator "`;`". In this case, these commands/pipelines are executed sequentially.

2.3 Parsing of a command pipeline

Pipes let you use the output of a program as the input of another one. A pipe operator is denoted by "`|`". Commands connected by one or more pipe operators form a pipeline. For example, with

```
cat os.txt | head -7 | tail -5
```

The last 5 lines of the first 7 lines (i.e., the lines 2-7) in the text file "os.txt" are output to terminal.

2.4 Parsing of command input and output redirection.

The command-line parser supports parsing of the following two redirection operators:

- >, which redirects `stdout` to a file following the operator
- <, which redirects `stdin` from a file following the operator

For example, "`ls -l > result.txt`" writes the result of "`ls -l`" to the file "result.txt" (instead of to the terminal).

2.5 Parsing of command background execution

If a command is terminated by the control operator "`&`", the shell executes the command asynchronously in the background.

2.6 Parsing of built-in "exit" command

When inputting "exit" to the command-line parser, the parser exits from execution.

3 Implementing your own shell terminal

Your job is to implement a shell program based on the above command-line parser. The sh program (in sh.c) already supports running basic commands. Your implementation need to support the following five additional features:

(1) Multiple commands in the same line.

(2) Command pipeline.

(3) Input and output redirection.

(4) Command background execution.

(5) Histroy

3.1 Multiple commands in the same line (10 points)

Multiple separate commands/pipelines can be put in the same line when delimited by the control operator "`;`". In this case, these commands/pipelines are executed sequentially.

3.2 Command pipeline (25 points)

In a command pipeline, each command is executed in its own process. You will need to use `pipe()` to pass the output of a command to the next command in the pipeline. You will also need to use `dup()` and `close()` to implement the standard input/output redirection for the pipeline.

3.3 Input and output redirection (20 points)

The shell should support the following two redirection operators:

- >, which redirects `stdout` to a file following the operator
- <, which redirects `stdin` from a file following the operator

For example, "`ls -l > result.txt`" writes the result of "`ls -l`" to the file "result.txt" (instead of to the terminal).

Related APIs: `open()`, `close()`, `dup()/dup2()`.

3.4 Command background execution (15 points)

If a command is terminated by the control operator "`&`", the shell executes the command asynchronously in the background. In other words, the shell does not wait for the completion of the command. Instead, it is ready for the next command immediately.

Since shell does not wait the background command to finish, it cannot reap the child process that was used to run the command in a synchronous way. In this assignment, a completed background command process should be reaped on the completion of the first foreground process after that background completes.

Hint: you may need to remember the PID of a background process and use `waitpid()` to reap it after it finishes.

3.5 Command History (30 points)

Modern shells don't just run the program you specify. They often have built in features like tab-completion and history tracking. In your implementation, you should add a built in function to your shell to track command history. Specifically, you should track the last 10 commands a user has entered and allow the user to re-execute these commands.

Add the built-in function *hist* to your shell. The *hist* function will perform two operations depending on passed arguments. If a user passes the string p̈rint(̈i.e the user types *hist print* into the shell), the last 10 commands the user typed will be printed and enumerated. If the user passes a number N (1¡N¡=10), the nth command will be re-executed.

Finally, make sure that single hist commands are not added to the history. If the hist command is part of a complicated multi-line command, it can be included in the history. See the test cases for examples.

3.6 Suggestion for implementing the above features

Through our past lectures and labs, you should already have the necessary knowledge to do the job. Here you may need to run the parser and study the base code to understand how the given parser parses a command line first. Because it will help you understand not only the basics of how a command is broken down and the tokens are remembered by the parser, but also some tricky implementation by the parser like how the parser handles a multi-command (i.e., three or more) pipeline.

3.7 The test cases

Here are the test cases that you may use to test your implementation of the shell program.

<u>**Test case 1**</u> (commandline pipeline): "`ls; ./sleep_and_echo öperating systems`¨" followed by "`ps`"

The expected output is explained as follows:

```
$ ls; ./sleep_and_echo "operating systems"
.              1 1 2048
..             1 1 2048
README         2 2 7
cat            2 3 16340
echo           2 4 15192
forktest       2 5 9512
grep           2 6 18560
init           2 7 15780
kill           2 8 15220
ln             2 9 15076
ls             2 10 17708
mkdir          2 11 15320
rm             2 12 15300
```

```
sh             2 13 31932
stressfs       2 14 16208
usertests      2 15 67320
wc             2 16 17072
zombie         2 17 14892
shutdown       2 18 14980
sleep_and_echo 2 19 15340
ps             2 20 14832
console        3 21 0
sne PID=7: "operating systems"
```

This test case runs a shell command and a user program in the same line. The "ps" should not indicate any zombie processes exist.

Test case 2 (commandline pipeline): "`cat README | wc`" followed by "`ps`"

The expected output is explained as follows:

```
$ cat README | wc
9 34 207
```

This pipeline consists of two commands. The first command outputs the content of the file "./README", which is served as the input of the second command. The second command prints the files 'word count' properties, specifically the number of lines, words, and characters the file contains in that order. The "ps" should not indicate any zombie processes exist.

Test case 3 (commandline pipeline): "`ls .  | grep s | wc`" followed by "`ps`"

The expected output is explained as follows:

```
$ ls . | grep s | wc
9 36 228
```

This is a pipeline consists of three commands. The last command (`wc`) outputs the word count properties of its input. The second command, (*grep s*) prints out the lines of its input which contain the character 's' at least once. This command in total prints the word count of each line of the *ls* command that contains an 's'. The "ps" should not indicate any zombie processes exist.

Test case 4 (input/output redirection): "echo operating system > aaa" followed by "`cat aaa`", and lastly "`ps`"

The expected output is explained as follows:

```
$ echo "Operating System" > aaa
$ cat aaa
"Operating System"
```

This command redirects the output of "`echo "operating systems"`" to file "./aaa". If the file "./aaa" does not exist, it is created. Otherwise, the output of `echo` overwrites the original content in the file. The last "ps" should not indicate any zombie processes exist. Hint: use the flags of "`O_CREAT|O_RDWR|O_TRUNC`" when opening the file "./aaa".

Test case 5 (input/output redirection): "`wc < README`" followed by "`ps`"

The expected output is explained as follows:

```
$ wc < README
9 34 207
```

This command redirects the input of "wc" to file "./README". Therefore, the expected output is README's entire content. The "ps" should not indicate any zombie processes exist.

Test case 6 (input/output redirection): "wc < README > testfile" followed by "cat testfile" and "ps"

The expected output is explained as follows:

```
$ wc < README > testfile
$ cat testfile
9 34 207
```

If you understand the previous two test cases, you should understand what's going on here. The last "ps" should not indicate any zombie processes exist.

Test case 7 (command background execution): "''./sleep_and_echo'' Hello world! &", the immediately "ls"; After the sleep_and_echo echos the message, do another ''ls'', followed by ''ps''

The expected output is explained as follows:

```
$ ./sleep_and_echo "Hello World!" &
$ ls
.               1 1 2048
..              1 1 2048
README          2 2 207
cat             2 3 16340
echo            2 4 15192
forktest        2 5 9512
grep            2 6 18560
init            2 7 15780
kill            2 8 15220
ln              2 9 15076
ls              2 10 17708
mkdir           2 11 15320
rm              2 12 15300
sh              2 13 31932
stressfs        2 14 16208
usertests       2 15 67320
wc              2 16 17072
zombie          2 17 14892
shutdown        2 18 14980
sleep_and_echo 2 19 15340
ps              2 20 14832
console         3 21 0
$ sne PID=5: "Hello World!"
ls
.               1 1 2048
```

```
..             1 1 2048
README         2 2 207
cat            2 3 16340
echo           2 4 15192
forktest       2 5 9512
grep           2 6 18560
init           2 7 15780
kill           2 8 15220
ln             2 9 15076
ls             2 10 17708
mkdir          2 11 15320
rm             2 12 15300
sh             2 13 31932
stressfs       2 14 16208
usertests      2 15 67320
wc             2 16 17072
zombie         2 17 14892
shutdown       2 18 14980
sleep_and_echo 2 19 15340
ps             2 20 14832
console        3 21 0
```

After "./sleep_and_echo Hello world! &" was entered, the next shell prompt prints immediately. So we can do the first "ls". After the `sleep_and_echo` program finishes (i.e., after "sne PID=854828: Hello world!" is printed), we can do another "ls". The (background) child process that ran the `sleep_and_echo` program was reaped when the second "ls" (which was the first foreground command after the background `sne` finished) was reaped. The last "`ps`" should not indicate any zombie processes exist.

Test case 8 (Command line history): "`type any series of commands`", then type "`hist print`" , and then "`ps`"

The expected output is explained as follows:

```
$ hist print
Previous command 1: ls
Previous command 2: ./sl./sleep_and_echo "Hello World!"
Previous command 3: cat testfile
Previous command 4: wc < README > testfile
Previous command 5: wc < README
Previous command 6: ls . | grep s | wc
Previous command 7: cat README | wc
Previous command 8: ls; ./sleep_and_echo "operating systems"
```

In this example, eight commands have been entered since the shell started. The shell tracked each command, and when "`hist print`" was entered, the shell prints each command, numbered in reverse chronological order. In this test case, command 1 was the last command typed in the terminal before "`hist print`" and command 8 was the first command tracked by the shell.

Test case 9 (Command line history): "`type a few more commands (`$N > 10$`)`", then type "`hist print`"

The expected output is explained as follows:

```
$ hist print
Previous command 1: ls
Previous command 2: ls
Previous command 3: ls
Previous command 4: ls
Previous command 5: ./sl./sleep_and_echo "Hello World!"
Previous command 6: cat testfile
Previous command 7: wc < README > testfile
Previous command 8: wc < README
Previous command 9: ls . | grep s | wc
Previous command 10: cat README | wc
```

following test case 8, a few additional commands were typed into the terminal. The shell tracked these commands and allowed some previously typed commands to be forgotten. In this case, the shell only maintains the last 10 commands typed.

Test case 10 (Command line history): "`hist print`", then type "`hist 1`", and finally "`hist print`"

The expected output is explained as follows:

```
$ hist print
Previous command 1: ls
Previous command 2: ls
Previous command 3: ls
Previous command 4: ls
Previous command 5: ./sl./sleep_and_echo "Hello World!"
Previous command 6: cat testfile
Previous command 7: wc < README > testfile
Previous command 8: wc < README
Previous command 9: ls . | grep s | wc
Previous command 10: cat README | wc
$ hist 1
.              1 1 2048
..             1 1 2048
README         2 2 207
cat            2 3 16340
echo           2 4 15192
forktest       2 5 9512
grep           2 6 18560
init           2 7 15780
kill           2 8 15220
ln             2 9 15076
ls             2 10 17708
mkdir          2 11 15320
rm             2 12 15300
sh             2 13 31932
stressfs       2 14 16208
usertests      2 15 67320
wc             2 16 17072
zombie         2 17 14892
shutdown       2 18 14980
sleep_and_echo 2 19 15340
ps             2 20 14832
console        3 21 0
```

```
testfile       2 22 10
$ hist print
Previous command 1: ls
Previous command 2: ls
Previous command 3: ls
Previous command 4: ls
Previous command 5: ./sl./sleep_and_echo "Hello World!"
Previous command 6: cat testfile
Previous command 7: wc < README > testfile
Previous command 8: wc < README
Previous command 9: ls . | grep s | wc
Previous command 10: cat README | wc
```

following test case 9, we can see that "`hist print`" was not tracked by our shell. Let's type that again, and then select one of the commands (in this case command 1) to re-run. Typeing '`hist 1`" should then rerun command 1 in '`hist print`" ('`ls`", in this case). After we rerun '`ls`", we should see that '`hist 1`" was not tracked as a command by the shell.

===== end of test cases =====

Last but not the least, your shell should be able to repeatedly run all the test cases correctly with one launch of the shell program.

4 Submit your work

Once your code is complete, please submit it on Brightspace by the deadline in a zip file.

Please also include a PROJ3.txt file explaining how you and your group members worked on this project. It is recommended that each group member work on at least one of the first 4 shell features, and that you collaborate on the history feature. Please comment on the work that each of you completed.

You can include the following info in this file:

- The status of your implementation (especially, if not fully complete).
- A description of how your code works, if that is not completely clear by reading the code (note that this should not be necessary, ideally your code should be self-documenting).
- Possibly a log of test cases which work and which don't work.
- Any other material you believe is relevant to the grading of your project.

Suggestion: Test your code thoroughly on a CS machine before submitting.

5 Grading

The following are the general grading guidelines for this and all future projects.

(1) The submission timestamp in Brightspace will be used to determine if your submission is on time or to calculate the number of late days. Please consult the syllabus for the courses late policy.

(2) If the submitted patch cannot successfully patched to the baseline source code, or the patched code does not compile:

```
1   TA will try to fix the problem (for no more than 3 minutes);
2   if (problem solved)
3     1%-10% points off (based on how complex the fix is, TA's discretion);
4   else
5     TA may contact the student by email or schedule a demo to fix the problem;
6     if (problem solved)
7       11%-20% points off (based on how complex the fix is, TA's discretion);
8     else
9       All points off;
```

So in the case that TA contacts you to fix a problem, please respond to TA's email promptly or show up at the demo appointment on time; otherwise the line 9 above will be effective.

(3) If the code is not working as required in the project spec, the TA should take points based on the assigned full points of the task and the actual problem.

(4) Lastly but not the least, stick to the collaboration policy stated in the syllabus: you may discuss with your fellow students, but code should absolutely be kept private. You may share code with your group members, but not with other students in the class. Any kind of cheating will result in zero point on the project, and further reporting.