CS 350 Operating Systems, Fall 2022
Programming Project 3 (PROJ3)

Out: 10/31/2022, MON
Due date: 11/16/2022, WED 23:59:59

In this project, you will implement your own shell terminal based on a command-line parser program provided. The goal of this project is to practice the various multiprocess programming handling and the corresponding API.

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 lab assignments) to complete this assignment.

2 The baseline code - a command-line parser program

This section introduces the command-line parser program (named "myshell"), which can be obtained by building the baseline code. 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.
- Parsing of a built-in "exit" command.

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. Your implementation need to support the following six features:

(1) Basic foreground command/program execution.

(2) Multiple commands in the same line.

(3) Command pipeline.

(4) Input and output redirection.

(5) Command background execution.

(6) `Ctrl-C` to terminate the current foreground command (instead of the shell program). (Bonus)

3.1 Basic foreground command/program execution (30 points)

When the shell program runs a command, it creates a child process to execute it. By default, the command is executed in the foreground, meaning the shell (i.e., the main process) waits for the completion of the command before displaying the next shell prompt and is ready for the next command. The shell (i.e., the main process) should reap the exited child process when the command finishes.

If the exit code of the command/program is non-zero, report the exit code (see test case 3 below).

Related process APIs: `fork()`, one of the `exec` functions, `wait()/waitpid()`, `exit()`.

3.2 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.

3.3 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()/dup2()` and `close()` to implement the standard input/output redirection for the pipeline.

3.4 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()`.

3.5 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.

3.6 `Ctrl-C` to terminate the current foreground command (10 bonus points)

With the shell command line parser in the base code, when hitting `Ctrl-C`, the shell parser is terminated due to the `SIGINT` signal. In your implementation, `Ctrl-C` should terminate the current foreground command if there is one, otherwise there is no effect of the `Ctrl-C`. To this end, you need to establish your own signal handler for the `SIGINT` signal. Remember to reap the terminated child process properly.

Related API: `signal()` (You will need to learn the concept of signals and the usage of the `signal()` function on your own.)

3.7 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.

3.8 The test cases

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

Test case 1 (basic foreground command/program execution): "`ls -a -l`" followed by "`ps`"

The expected output is explained as follows:

```
myshell> ls -a -l
total 1572
drwxr-xr-x  2 jboubin jboubin   4096 Sep 16 08:24 .
drwxr-xr-x 10 jboubin jboubin   4096 Sep 15 22:16 ..
-rw-r--r--  1 jboubin jboubin     16 Sep 15 22:17 .gitignore
-rw-r--r--  1 jboubin jboubin  11424 Sep 15 22:17 main.c
-rw-r-----  1 jboubin jboubin  13664 Sep 16 08:24 main.o
-rw-r--r--  1 jboubin jboubin    305 Sep 15 22:17 Makefile
```

```
-rwxr-x---  1 jboubin jboubin 775912 Sep 16 08:24 myshell
-rw-r--r--  1 jboubin jboubin   7604 Sep 15 22:17 parser.c
-rw-r--r--  1 jboubin jboubin   2471 Sep 15 22:17 parser.h
-rw-r-----  1 jboubin jboubin   8648 Sep 16 08:24 parser.o
-rw-r--r--  1 jboubin jboubin    368 Sep 15 22:17 sleep_and_echo.c
-rwxr-x---  1 jboubin jboubin 755840 Sep 16 08:24 sne
myshell>
```

The test case first runs the `ls` command with the `-a` and `-l` options. A child process was created to run the command. Then running the "ps" command, which desplays the currently-running processes, would give you something like:

```
myshell> ps
PID TTY          TIME CMD
4064571 pts/0    00:00:00 bash
4064784 pts/0    00:00:00 myshell
4065087 pts/0    00:00:00 ps
myshell>
```

If child process reaping is not properly done by the parent, you will see someting like:

```
myshell> ps
PID TTY          TIME CMD
949917 pts/1    00:00:00 bash
950620 pts/1    00:00:00 myshell
950625 pts/1    00:00:00 ls <defunct>
950635 pts/1    00:00:00 ps
```

where "ls <defunct>" means there is a zombie "ls" process.

Test case 2 (basic foreground command/program execution): "`./sne I am testing my shell program`" followed by "`ps`"

The expected output is explained as follows:

```
myshell> ./sne I am testing my shell program
sne PID=4066935: I am testing my shell program     <-- this prints after 3 secs
myshell>
```

This test case runs the `sne` user program that was placed in the same folder as the shell program. The `sne` (sleep and echo) program sleeps for 3 seconds, prints its own PID, and echoes whatever message provided by the user. The source code of the `sne` program has been provided. Just run "make" to compile and get the program. You can change the duration of the sleep by changing the "sleep_seconds" variable in the makefile.

Running the "`ps`" command should not indicate any zombie processes exist.

Test case 3 (basic foreground command/program execution): first change the value of the "ecode" variable in the Makefile to 10 and "`make`", then do "`./sne hi there`" followed by "`ps`".

The expected output is explained as follows:

```
$ vi Makefile    <-- Change the ecode variable value to 10 here
$ make           <-- Rebuild the project
gcc -static -DSECS=3 -DECODE=10 sleep_and_echo.c -o sne
```

```
$ ./myshell     <-- Run myshell
myshell> ./sne hi there   <-- Run the sne program
sne PID=864267: hi there  <-- Output of the sne program

Non-zero exit code (10) detected  <-- sne's non-zero exit code is reported
myshell>
```

Running the "ps" command should not indicate any zombie processes exist.

Test case 4 (multiple commands in the same line): First change the value of the "ecode" variable in Makefile back to 0, rebuild the project. Then do "`ls -a -l; ./sne operating systems`" followed by "`ps`"

The expected output is explained as follows:

```
myshell> ls -a -l; ./sne operating systems
total 1572
drwxr-xr-x  2 jboubin jboubin   4096 Sep 16 08:24 .
drwxr-xr-x 10 jboubin jboubin   4096 Sep 15 22:16 ..
-rw-r--r--  1 jboubin jboubin     16 Sep 15 22:17 .gitignore
-rw-r--r--  1 jboubin jboubin  11424 Sep 15 22:17 main.c
-rw-r-----  1 jboubin jboubin  13664 Sep 16 08:24 main.o
-rw-r--r--  1 jboubin jboubin    305 Sep 15 22:17 Makefile
-rwxr-x---  1 jboubin jboubin 775912 Sep 16 08:24 myshell
-rw-r--r--  1 jboubin jboubin   7604 Sep 15 22:17 parser.c
-rw-r--r--  1 jboubin jboubin   2471 Sep 15 22:17 parser.h
-rw-r-----  1 jboubin jboubin   8648 Sep 16 08:24 parser.o
-rw-r--r--  1 jboubin jboubin    368 Sep 15 22:17 sleep_and_echo.c
-rwxr-x---  1 jboubin jboubin 755840 Sep 16 08:24 sne
sne PID=4068836: operating systems
myshell>
```

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

Test case 5 (commandline pipeline): "`cat ./Makefile | head -1`" followed by "`ps`"

The expected output is explained as follows:

```
myshell> cat ./Makefile | head -1
sleep_seconds=3
myshell>
```

This pipeline consists of two commands. The first command outputs the content of the file "./Makefile", which is served as the input of the second command. The second command prints the first line of it's input (i.e., the first line in the file "./Makefile"). The "ps" should not indicate any zombie processes exist.

Test case 6 (commandline pipeline): "`cat ./Makefile | head -1 | wc -c`" followed by "`ps`"

The expected output is explained as follows:

```
myshell> cat ./Makefile | head -1 | wc -c
```

```
    16
    myshell>
```

This is a pipeline consists of three commands. The last command (`wc -c`) outputs the number of characters in its input. The "`ps`" should not indicate any zombie processes exist.

Test case 7 (input/output redirection): "echo operating system > aaa" followed by "`cat aaa`", and lastly "`ps`"

The expected output is explained as follows:

```
    myshell> echo operating systems > aaa
    myshell> cat aaa
    operating systems
    myshell>
```

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 8 (input/output redirection): "`head -1 < Makefile`" followed by "`ps`"

The expected output is explained as follows:

```
    myshell> head -1 < Makefile
    sleep_seconds=3
    myshell>
```

This command redirects the input of "`head -1`" from file "./Makefile". Therefore, the expected output is the first line of Makefile's content. The "`ps`" should not indicate any zombie processes exist.

Test case 9 (input/output redirection): "`head -1 < Makefile > aaa`" followed by "`cat aaa`" and "`ps`"

The expected output is explained as follows:

```
    myshell> head -1 < Makefile > aaa
    myshell> cat aaa
    sleep_seconds=3
    myshell>
```

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 10 (command background execution): "`./sne Hello world!  &`", the immediately "`ls`"; After the `sne` echos the message, do another "`ls`", followed by "`ps`"

The expected output is explained as follows:

```
myshell> ./sne Hello world! &
myshell> ls
```

```
main.c  main.o  Makefile  myshell   parser.c  parser.h  parser.o  sleep_and_echo.c  sne
myshell> sne PID=854828: Hello world!  <-- This line prints 3 secs after the sne execution

myshell> ls
main.c  main.o  Makefile  myshell   parser.c  parser.h  parser.o  sleep_and_echo.c  sne
myshell> ps
PID TTY          TIME CMD
851692 pts/1    00:00:00 bash
851756 pts/1    00:00:00 myshell
854891 pts/1    00:00:00 ps
myshell>
```

After "./sne Hello world! &" was entered, the next shell prompt prints immediately. So we can do the first "ls". After the `sne` program finishes (i.e., after "sne PID=854828: Hello world!" is printed), we can do another "ls". The (background) child process that ran the `sne` 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.

<u>Test case 11</u> (Ctrl-C to terminate curret foreground command/program): "`./sne hello world`", then immediately "`Ctrl-C`", then "`ps`"

The expected output is explained as follows:

```
    myshell> ./sne hello world
    ^Cmyshell> ps   <--^C is when Ctrl-C is presssed
    PID TTY          TIME CMD
    851692 pts/1    00:00:00 bash
    866213 pts/1    00:00:00 myshell
    866255 pts/1    00:00:00 ps
    myshell>
```

The foreground process in the shell is interrupted, and is properly reaped (i.e., no zombie process is in the output of "`ps`").

<u>Test case 12</u> (Ctrl-C to terminate curret foreground command/program): Just do `Ctrl-C` without any foregroud process running

The expected output is explained as follows:

```
    myshell> ^C  <-- Ctrl-C without any foreground process running
    Ctrl-C catched. But currently there is no foreground process running.
    myshell>
```

In this case, instead of terminating the myshell process, report Ctrl-C signal catched and do nothing.
===== end of test cases =====
You may also try the above test cases with the sample myshell program (i.e., the "myshell-sample" program in brightspace), which is implemented by the instructor, see what the expected outputs are (Do "`chmod ugo+x myshell-sample`" if there is no execution permission to the program).

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

4 Submit your work

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

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 you fellow students, but code should absolutely be kept private. Any kind of cheating will result in a zero on the project and further reporting.