CPSC 3500 Computing Systems

Project 2: A Simple Shell Using Multiprocessing and Pipes

16 points

Assigned: Friday, 01/21/2022

Due: 11:59PM, Thursday, 02/03/2022

Note: Students may choose either individual or group projects on this assignment. If you had a hard time on project 1, it is highly recommended that you team with someone. For a group project, the group size is capped at 3 members and only one submission is required.

1. Background

A shell is a command-line interpreter that provides a command-line user interface for UNIX-like operating systems. The shell is just another program. For example, the Bash shell is an executable named `bash` that is usually located in the `/bin` directory, namely, `/bin/bash`.

2. Assignment Requirement

Write a simple shell in C/C++. The requirements are as follows.

2.1 The Simple Shell Handling One Command

Your shell executable should be named `myshell`. Your shell source code should be mainly in `myshell.c` or `myshell.cpp`, but you are free to add additional source code files as long as your `Makefile` works, compiles and generates an executable named `myshell` in the same top level directory as the `Makefile`. If we cannot run `make` and then `myshell`, your assignment submission receives zero point.

Your shell does not have to run continuously. The shell starts with displaying a prompt and then waits for input. The prompt should be exactly '`myshell$`'. No space, no extra characters. Example with a command:

```
myshell$echo hello world!
```

You shell should exit after the command is executed.

Your shell should read a line from stdin (file descriptor 0). You may assume that the entire command is on one line. This line should be parsed out into *command name* and *all its arguments*. In other words, tokenize it. You may assume that the only supported delimiter is the whitespace character (ASCII character number 32). You may assume that a command won't have more than 20 tokens and each token length won't exceed 20.

The above command `echo hello world!` should be tokenized into three words:

```
"echo", "hello", "world!"
```

After parsing and lexing the command, your shell should execute it. In this part, just focus on running one command. Also, executing a command is done by using `fork()` to create a child process that in turn invokes `execvp()` on the command.

You **CANNOT** use `system()`, as it just invokes the `/bin/sh` shell to do all the work.

The parent should wait for the child process to terminate before exiting. The parent should wait on termination of its child and then print the child's PID and exit status. The message should be exactly: "process [PID] exits with [exit status value]".

2.2 Run a Pipeline of Commands using Pipes

One of the defining aspects of the UNIX programming environment is the *tools* concept: a lot of little programs that do one thing but can be combined together. The act of combining programs means that the output of one program is used as the input to another program. The mechanism that accomplishes this is the **pipe**. In the shell, it is denoted by a vertical bar `|`.

For example, the `ls` command lists all the files in a directory. The `wc` command counts lines, words, and characters. The number of files and directories in my current directory is:

```
ls | wc -l
```

To find the top ten words in Herman Melville's *Moby Dick* and a count of their occurrence, we can run:

```
cat moby.txt | tr A-Z a-z | tr -C a-z \n |sed  /^$/d | sort | uniq -c | sort -nr | sed 10q
```

Where:

- `cat moby.txt` puts the contents of the file `moby.txt` onto the standard output stream (e.g., file descriptor 1).
- `tr A-Z a-z` translates all uppercase letters to their corresponding lowercase ones. This ensures that mixed-case words all get converted to lower case and will therefore look the same when we compare them.
- `tr -C a-z \n` translates anything that is not a lowercase letter into a newline (\n). This gives us one word per line as well as a lot of extra blank lines.
- `sed /^$/d` runs `sed`, the stream editor, to delete all empty lines (i.e., lines that match the regular expression `^$`).
- `sort` sorts the output. Now all of our words are sorted.
- `uniq -c` combines adjacent lines (words) that are identical and writes out a count of the number of duplicates followed by the line.
- `sort -nr` sorts the output of `uniq` by the count of duplicate lines. The sorting is in inverse numeric order (`-nr`). The output from this is a frequency-ordered list of unique words.
- `sed 10q` tells the stream editor to quit after reading ten lines. The user will see the top ten lines of the output of `sort -nr`.

The pipe operator |, allows us to redirect outputs (originally destined to `stdout`, file descriptor 1) of a process to the `stdin` (file descriptor 0) of another process. In the above example, we connect `stdout` of a command to `stdin` of its immediately following command. As a result, the outputs of a command's execution are redirected to the inputs of the next command's execution.

Augment your shell (implemented in Section 2.1) to be capable of executing a sequence of programs that communicate through a pipe. For example, if the user types `ls | wc -l`, your shell should fork two child processes that each executes one command (by `execvp()`), which together will calculate the number of files and directories in the current directory.

While this example shows two processes communicating through a pipe, your shell should support pipes between more than two processes.

2.3 Additional Requirement

Error handling on a child process. If the child fails to execute the desired command using `execvp()`, it should print an error message using `perror()` and exit with a code of **1**. Otherwise, it should exit with a code of **0**.

The parent process should not exit until all its children terminate. The parent process should wait on termination of all its child processes and display the proper messages each containing a child process' PID and exit status as specified in Section 2.1. Do not be concerned if some of the messages are interspersed with some output from the commands themselves.

Your submission should include a Makefile and README. The included Makefile controls the compilation of your code and produces the shell executable. The README should state strength and weakness of your shell. If it is a group project, the README should list team members and their respective contributions.

Check return values of all system calls. Your code should check return values of all system calls including `fork, execvp, wait, dup2, pipe, exit`, etc.

Your code should not have obvious memory leak.

Your code should be well-commented. You do not have to comment every single line. Important segments of code should be commented for being readable.

Your shell should not produce any debugging and testing messages. I understand that you may find debugging and testing messages helpful at some point. But be sure to turn them off in your assignment submission. Otherwise, penalty can be imposed on those unsolicited messages.

The program should be compiled and executed. If I cannot compile your code with the provided Makefile, your submission receives zero point. If the program immediately aborts abnormally once it is started, your submission receives zero point.

2.4 Summary

The shell should be able to ***run one command*** as well as ***a pipeline of commands***. When each child process terminates, the shell must print the exit status together with PID for the child process. To make it simple, your shell can simply exit after all the child processes are finished. You may assume that the number of commands won't be more than 10, the number of arguments for each command won't go beyond 20 and the length of command names or arguments won't exceed 20.

The goal of this assignment is to further your understanding in multiprocessing and inter-process communication via pipes. A set of system calls including *fork*, *execvp*, *wait*, *pipe*, *dup2* and *exit* will be used.

You do not have to implement multi-line commands, environment variables, or I/O redirection.

Don't be intimidated by the length of this write-up. My version of this project is well under 300 lines of code even with additional functionalities, a fair number of comments and blank lines.

3. Tips

3.1 System calls

Many details regarding system calls (e.g., `fork`, `execvp`, `dup2`, ...) can be found in the system manual pages; type in `man function` in `cs1.seattleu.edu` to get more information about function. If `function` is a system call, `man 2 function` or `man 3 function` can ensure that you receive the correct man page, rather than one for a system utility of the same name. For example, if you want to learn more about `dup2`, you can type in: `man 2 dup2`

Some system calls require `man 3`. Taking `execvp` as an example, you have to type in: `man 3 execvp`

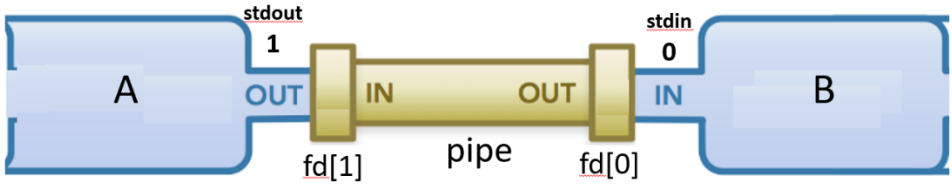
Tons of online sources are also available. For instance, if you want to learn more about `dup2`, google "`Linux dup2`".

3.2 Pipes

A pipe is a unidirectional communication channel with data flowing into one end and flowing out from the other end. If process *A* wants to send its outputs (that originally go to `stdout`) to process *B* which wants to receive *A*'s outputs instead of data supposedly coming from `stdin`, we can create a pipe between them via system call `pipe`. Then, *A* redirects its outputs to one end of the pipe. *B* receives inputs from the other end of the pipe rather than from `stdin`.

Parent process

```
int  fd[2]; //file descriptors for a pipe
pipe(fd);   //create a pipe
```



Child process A

```
dup2(fd[1], 1);
close(fd[1]);
close(fd[0]);
```

Child process B

```
dup2(fd[0], 0);
close(fd[1]);
close(fd[0]);
```

Consider a shell with two commands connected by a pipe as shown in the graph above. The parent process creates the pipe, returning two file descriptors: `fd[0]` and `fd[1]`, which are used to receive data and to send data via the pipe respectively. Then, the parent forks two child processes *A* and *B* corresponding the two commands respectively. Now, all the three processes have the two file descriptors

open. Why? Recall what happens to open files when a child process is created. [Open file tables are duplicated for fork()]

Process *A* uses dup2 to redirect its stdout to fd[1]. The system call dup2 atomically closes file descriptor 1 and duplicates opened file fd[1] to file descriptor 1. Consequently, file descriptors fd[1] and 1 both are associated with the same open file. *A* needs to close unused file descriptors, that is, fd[1] and fd[0]. Similarly, process *B* redirects its stdin to the pipe fd[0] and closes all unused file descriptors. Remember that *A* by default sends outputs to file descriptor 1 and that *B* by default reads inputs from file descriptor 0. [Read this paragraph carefully!!!]

After the two child processes complete the aforementioned operations, *A*'s outputs (originally going to stdout) will flow to *B*'s stdin through the pipe until *A* shuts down the pipe normally or abnormally.

Of course, the parent process is not involved with data transferring between *A* and *B*. It should close the two file descriptors fd[0] and fd[1], say, after the two child processes are created. Why?

The above example just shows one pipe for two commands. You need to ask yourself a question: what if there are more than 2 commands? (1) How many pipes should be created? (2) Where are those pipes stored for easy/convenient accesses?

3.3 Reading a Line

fgets is a fine function to use for reading a line. ***Don't*** use *gets* since it doesn't allow you to specify the size of the buffer and is a target for buffer overflow attacks. If you use C++ *iostream* cin, ***then*** *getline()* could be a help.

3.4 Token Storage and Argument List for execvp

There are many ways to store tokens for a pipeline of commands. One possible solution is to use C-strings, namely, a 2-dimensional array of characters. For example, we can use the following 2-dimensinal array to store the tokens for up to 10 commands.

```
char buf[10][500];
```

Each buf[i] is a C-string, storing a command's tokens. (each token ends with a null character '\0')

The table below shows token storage for a pipeline of two commands:

```
ls -laF / | tr a-z  A-Z
```

Index	0	1	2	3	4	5	6	7	8	9	10	…
buf[0]	'l'	's'	'\0'	'-'	'l'	'a'	'F'	'\0'	'/'	'\0'		
buf[1]	't'	'r'	'\0'	'a'	'-'	'z'	'\0'	'A'	'-'	'Z'	'\0'	
…												

Given this token storage as shown above, how does a child process construct the argument list for execvp?

Consider the child process running the command ls -laF /.

Below is the sample code for the child process to construct the argument list and run the command.

```
//argument list for execvp()
char* args[50];

//Now a child process wants to execute its command.
//It has to pack the tokens into argument list as required by execvp

int pos = 0;
for (int i = 0; i < 3; i++) {
    args[i] = (char*)&buf[0][pos];
    pos += strlen(args[i]) + 1; //point to next token
}
args[3] = (char*)NULL; //very important to end the argument list!!!!

//execute the command
execvp(args[0], args);
```

I do not claim that this is the only solution, but I hope many of you will find it helpful to understand calling execvp(). If you did well in CPSC 2430 Data Structures, you can definitely find better solutions. Moreover, 3 here is hard-coded. The above code is mainly for illustration.

3.5 C-string vs. C++ std:string

C-string and C++ std:string are different! Many students in my prior classes had difficulties understanding their differences (which should be learned in their prior programming course sequences). They tended to use them interchangeably without cautions. Unfortunately this was one of root causes for segmentation faults. Please be advised that they should be handled differently. You should cram for these two instead of simply coming to me for help.

3.6 Closing Unused File Descriptors

After a child process is forked, the child process has the same set of the open files as its parent process (as discussed in class, all open files are open for each child process). Each process should close all unused file descriptors. It is crucial that the parent process closes the file descriptors of the pipes (before wait() on any child process). Otherwise, any processes reading from the pipes will never exit.

3.7 Makefile

You can revise the Makefile in project 1 for project 2.

Please refer to http://mrbook.org/blog/tutorials/make/ for further details.

3.8 Important Questions to Answer Before Coding

Before starting to write your code, there are a few questions you need to answer:

Consider a shell receiving a pipeline of N (N>1) commands.

- How many pipes should be created?
- How many child processes should be created?
- Where should the pipes be created, in parent or child process?

- When should the pipes be created, before or after creating child processes?
- Which pipe(s) should a child process use given multiple pipes? In other words, how do you assign pipes to child processes?

3.9 Last Note

In this project, you can assume that all commands are properly formatted and correct, to simplify your code.

4. Testing

4.1 Kill a Hung Process

In multi-processing programming, bugs seem to be complicated and daunting. Often, your process will hang up. If that happens, please do not panic. The first thing to do is killing the hung process. Assume the process is `myshell`.

Use `ps` to display the process info

```
$ ps
```

You will probably see something like this:

```
PID TTY          TIME CMD
28998 pts/2    00:00:00 bash
31052 pts/2    00:00:00 myshell
31055 pts/2    00:00:00 myshell
31056 pts/2    00:00:00 ps
```

Use `kill` to kill all hung processes (`kill -9 pid`)

```
$ kill -9 31052
$ kill -9 31055
```

You will see something like this:

```
[1]-  Killed      ./myshell
[2]+  Killed      ./myshell
```

4.2 Single Command Test Case

Command	Messages displayed by your shell (PID could be different)

`echo hello, world`	`hello, world`
	
	`process 16589 exits with 0`

`echo I am a SeattleU CS student`	`I am a SeattleU CS student` `process 16589 exits with 0`

`ls -l`	`total 100`
	`-rwxrwx--- 1 zhuy zhuy 10428 Mar 27 08:44 A`
	`-rw-rw---- 1 zhuy zhuy 1007 Mar 27 08:44 a.cpp`
	`-rwxrwx--- 1 zhuy zhuy 55797 Apr 3 21:06 proj`
	`-rw-rw---- 1 zhuy zhuy 6070 Apr 3 21:06 proj1.cpp`
	`Process 16146 exits with 0`

4.3 A Pipeline of two commands

```
ls -laF / | tr a-z A-Z
```

The displayed messages of the shell should be like:

```
TOTAL 224
DRWXR-XR-X  25 ROOT ROOT  4096 MAR 29 19:56 ./
DRWXR-XR-X  25 ROOT ROOT  4096 MAR 29 19:56 ../
-RW-R--R--   1 ROOT ROOT     0 MAR 29 19:56 .AUTOFSCK
DRWXR-XR-X   5 ROOT ROOT  4096 APR  1  2013 BACKUP/
DRWXR-XR-X   2 ROOT ROOT  4096 MAR 30 04:07 BIN/
DRWXR-XR-X   4 ROOT ROOT  4096 MAR 29 19:41 BOOT/
DRWXR-XR-X  11 ROOT ROOT  4020 MAR 29 19:56 DEV/
DRWXR-XR-X 109 ROOT ROOT 12288 APR  2 17:19 ETC/
DRWXR-XR-X   6 ROOT ROOT  4096 JAN  9  2012 HOME/
DRWXR-XR-X  11 ROOT ROOT  4096 MAR 30 04:07 LIB/
DRWXR-XR-X   8 ROOT ROOT 12288 MAR 30 04:07 LIB64/
DRWX------   2 ROOT ROOT 16384 JUN 14  2010 LOST+FOUND/
DRWXR-XR-X   2 ROOT ROOT  4096 OCT  1  2009 MEDIA/
DRWXR-XR-X   2 ROOT ROOT  4096 JUL 10  2014 MISC/
DRWXR-XR-X   2 ROOT ROOT  4096 OCT  1  2009 MNT/
DRWXR-XR-X   2 ROOT ROOT  4096 OCT  1  2009 OPT/
DR-XR-XR-X 369 ROOT ROOT     0 MAR 29 19:55 PROC/
-RW-------   1 ROOT ROOT  1024 JUN 14  2010 .RND
DRWXR-X---  21 ROOT ROOT  4096 JAN  8 11:58 ROOT/
DRWXR-XR-X   2 ROOT ROOT 12288 MAR 30 04:07 SBIN/
DRWXR-XR-X   4 ROOT ROOT     0 MAR 29 19:55 SELINUX/
Process 16345 exits with 0
DRWXR-XR-X   2 ROOT ROOT  4096 OCT  1  2009 SRV/
DRWXR-XR-X  11 ROOT ROOT     0 MAR 29 19:55 SYS/
DRWXR-XR-X   3 ROOT ROOT  4096 JUN 14  2010 TFTPBOOT/
DRWXRWXRWT   8 ROOT ROOT 20480 APR  4 14:59 TMP/
DRWXR-XR-X  16 ROOT ROOT  4096 JUN 14  2010 USR/
DRWXR-XR-X  25 ROOT ROOT  4096 JUN 14  2010 VAR/
Process 16346 exits with 0
```

4.4 A Pipeline of Three Commands

```
ls -alF / | grep bin | cat -n
```

The displayed messages of the shell should be like:

```
Process 16480 exits with 0
Process 16481 exits with 0
1  drwxr-xr-x   2 root root  4096 Mar 30 04:07 bin/
```

```
2  drwxr-xr-x   2 root root 12288 Mar 30 04:07 sbin/
Process 16482 exits with 0
```

4.5 A Pipeline of Four Commands

```
ls -alF  /  | grep bin | tr a-z A-Z | rev  |  cat -n
```

The displayed messages of the shell should be like:

```
Process 16554 exits with 0
Process 16555 exits with 0
Process 16556 exits with 0
1  /NIB 70:40 03 RAM 6904  TOOR TOOR 2   X-RX-RXWRD
2  /NIBS 70:40 03 RAM 88221 TOOR TOOR 2   X-RX-RXWRD
Process 16557 exits with 0
Process 16558 exits with 0
```

4.6 A Pipeline of Many Commands with More Data

```
cat myshell.cpp | tr A-Z a-z  | tr -C a-z  \n | sed  /^$/d | sort | uniq -c  |  sort -nr |  sed 10q
```

The displayed messages of the shell should be like:

```
Process 16940 exits with 0
Process 16941 exits with 0
Process 16942 exits with 0
Process 16943 exits with 0
Process 16944 exits with 0
Process 16945 exits with 0
Process 16946 exits with 0
     51 i
     34 cmd
     30 if
     24 token
     24 int
     22 flag
     17 return
     17 g
     16 else
     14 vs
Process 16947 exits with 0
```

5. Submission

Before submission, you should make sure that your code has been compiled and executed on `cs1.seattleu.edu` and your submission include all required files!

The following files must be included in your submission:

- `README`
- `*.h: all .h files`
- `*.c/cpp: all .c/cpp files`
- `Makefile`

You should create a package `p2.tar` including the required files as specified above, by running the command:

```
tar  -cvf   p2.tar  README  *.h  *.c  Makefile
```

Then, use the following command to submit p1.tar:

```
/home/fac/zhuy/zhuy/class/SubmitHW/submit3500   p2   p2.tar
```

If submission succeeds, you will see a message similar to the following one on your screen:

```
======================Copyright(C)Yingwu Zhu=========================
Mon Jan 13 12:43:34 PST 2020
Welcome testzhuy!
You are submitting hello.cpp for assignment p2.
Transferring file.....
Congrats! You have successfully submitted your assigment! Thank you!
Email: zhuy@seattleu.edu
=====================================================================
```

You can submit your assignment multiple times before the deadline. Only the most recent copy is saved.

The assignment submission will close automatically once the deadline passes. You need to contact me for instructions on your late submission. Do NOT email me your submission!

For a group project, only one submission is required.

6. Grading Criteria

Label	Notes
a. Meeting submission requirements (2 pts)	• Your Makefile works properly. • All required files are included in your submission. • If incomplete submission prevents your program from being executed, you may receive zero point on this assignment.
b. README & well-commented code (2 pts)	• README contains the required items (strengths & weakness) • Code is commented so that your code is easy to follow
c. Functionality (10 pts)	The shell should behave as specified. • Working with one command • Working with one pipe • Working with two pipes • Working with more than two pipes

	• Working with multiple pipes & a lot of data
d. Resource management & Outputs (2 pts)	• Check return values for system calls. • No obvious memory leaks (if applicable). • Each process closes unused file descriptors • Display messages properly. • No debugging/testing messages • No unsolicited messages scrambling on screen
e. Overriding policy	If the code cannot be compiled or can barely executed (segmentation faults once it starts, for instance), it results in zero point on this assignment.
f. Late submission	Please refer to the late submission policy on Syllabus.