CPSC 3500 Computing Systems

Project 1: Implementing a Simple Shell Using Multi-Processing

Assigned: 2:05PM, Wednesday, 01/16/2019

Due: 11:59PM, Wednesday, 01/23/2019

Introduction

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:

1. `cat moby.txt` puts the contents of the file `moby.txt` onto the standard output stream.
2. `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.
3. `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.
4. `sed '/^$/d'` runs `sed`, the stream editor, to delete all empty lines (i.e., lines that match the regular expression `^$`).
5. `sort` sorts the output. Now all of our words are sorted.
6. `uniq -c` combines adjacent lines (words) that are identical and writes out a count of the number of duplicates followed by the line.
7. `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.
8. `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`.

In the shell, four kernel mechanisms are crucial to accomplishing the above functionality:

fork *http://man7.org/linux/man-pages/man2/fork.2.html*
The *fork* system call is used to create new processes. Each command in the pipeline has to run as a separate process (we do not consider built-in commands here, such as `cd` and `exit`). The result of a *fork* is a cloned copy of the parent process.

execvp *http://man7.org/linux/man-pages/man3/exec.3.html*
The *execvp* system call overlays a new program onto the current process. Without execvp, we would never be able to run new programs. When the shell runs a command, it forks itself and the child then loads and runs the new program via *execvp*.

dup2 *http://man7.org/linux/man-pages/man2/dup.2.html*
dup2 simply duplicates one open file descriptor (the handle that is used to read and write files) onto another file descriptor number. This is the basis of I/O redirection. Under UNIX, programs that may interact with the terminal expect to have three file descriptors open and ready for use. File descriptor 0 is the standard input, file descriptor 1 is the standard output, and file descriptor 2 is the standard error. Typically, standard input comes from the keyboard and standard output and error go to the virtual terminal (terminal window running the shell). However, any or all of these can be redirected to other files (or devices; there is usually no distinction in UNIX). As an example of how this is done, consider redirecting the standard output of a program. Before calling *execvp* the child process opens the desired output file. The *open* system call returns a file descriptor, which is a small integer. The child then calls *dup2* to duplicate this new file descriptor onto file descriptor 1, the standard output. *dup2* causes the current file descriptor 1 to be closed and the new one is duplicated onto 1. All further output to file descriptor 1 (standard output) will now go to this newly-opened file. After this, the child process calls *execvp* to run the desired program, which will simply write to file descriptor 1 for its output.

pipe *http://man7.org/linux/man-pages/man2/pipe.2.html*
The *pipe* system call sets up a unidirectional communication channel using file descriptors. It creates two file descriptors: any data written to the second file descriptor will be read from the first file descriptor. Coupled with *fork*, *execvp*, and *dup2*, it allows one process to have a communication stream to another process.

Objectives

Your assignment will be implementing a simple shell that will ***run one command*** or ***a pipeline of commands***. When each process terminates, the shell will print the exit status together with PID for the process. To make it simple, your shell can simply exit after the command or pipeline of command is executed.

Your goal in this assignment is to become familiar with the basic set of system calls that let you create processes, establish pipes between them, and detect when a child process has died. This assignment will use the *fork*, *execvp*, *wait*, *pipe*, *dup2*, and *exit* system calls.

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, including a fair number of comments and blank lines.

Group size

You can form a group of 2 students to complete this project. You may also take it as an individual project (but no bonus for taking it as an individual project)

Languages

The assignment must be done in C or C++. The program must be compiled and submitted on cs1.seattleu.edu. You will receive no credit if your program cannot pass the compiler or execute.

Specifications

Your assignment is to write a rudimentary shell, or command interpreter, with very specific capabilities. Here is what it will do, step by step:

1. The standard input is coming from a terminal, print a prompt "`MyShell$ `"
2. Get a command line from the user. You may assume that the entire command is on one line. This line may be a pipeline of one or more commands.
3. Parse the command line. Each command is a sequence of one or more tokens separated by whitespace (spaces or tabs). Tokens containing spaces must be quoted with either single or double quotes. Commands may be piped together by separating them with a vertical bar (|). There need not be a space on either side of the vertical bar. Examples of valid input lines are:

   ```
   echo I "am a command"
   ls|wc -l
   "ls" | wc '-l'
   ```

   The last two commands are identical.

   Break the command and arguments into an argument list (an array of char *). You may assume that no command will have more than 50 arguments. You may also assume that there are no more than 10 commands in this project.

   The result of your parsing will be a list of piped commands. Each command will be parsed into a list of arguments.

4. Execute the pipeline of commands. If one command has its output going to another command, then create a pipe and set the standard output (file descriptor of 1) of the command to the pipe. If the command is reading input from a pipe then set the standard input (file descriptor of 0) to the pipe. See the *Hints* section for more advice.

   Execute each command using *fork* and *execvp*. You should **not** use the *system* library function. If you do, your grade for the assignment will be **0**. If the child cannot execute the desired command (execvp fails), then print an error message (see *perror*) and exit the child with an exit code of **1**.

5. The parent will now wait for all the child processes to terminate before exiting. To do this, loop on *wait* until *wait* returns a value of -1. For each process that terminates, print that child's PID and

exit status. Do not be concerned if some of the termination messages are interspersed with some output from the commands themselves.

Hints

Develop and test incrementally!

Develop and test your code incrementally. Write a 10-30 lines of code at a time and then test it. For example, try this sequence:

1. Prompt for a command, get the line, print it, and repeat.
2. Parse the command line into tokens. A token is either a quoted string, a set of non-space characters, or a `|` character. Test this extensively to ensure that this is robust. If a string starts with a single quote, it must end with a single quote. If it starts with a double quote (`"`) then it must end with a double quote. For example, the command:

   ```
   'abc'   "de f'g"  hij|  k "lm | no"
   ```

   must parse into the tokens:

   ```
   token 1: "abc"
   token 2: "de f'g"
   token 3: "hij"
   token 4: "|"
   token 5: "k"
   token 6: "lm | no"
   ```

3. Build up a list of commands. Each command is a sequence of arguments. A pipe separates commands. From the above example, the list will be:

   ```
   command 1: "abc", "de f'g", "hij"
   command 2: "k", "lm | no"
   ```

4. Set up a pipe (if necessary) and execute a command using *fork*, *dup2*, and *execvp*.
5. Loop and wait for all processes to die, printing their status. Make sure that all processes do indeed terminate. **Also do not forget close all the pipes in your parent process!**

Malloc and strcpy

If you use *malloc* for dynamic memory allocation, be sure to use *free* to clean it up. Your program does not need to use any dynamic memory allocation or string copying for parsing the command line (although you certainly may do this). All you need to do is keep track of pointers to the start of each token and terminate the end of the token with a 0 (or NULL) within the command line buffer. Unnecessary memory allocation and copying is your enemy!

Getting lines

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.

Creating an argument list

Your parsing function will split a line containing a command into an array of pointers to characters.

You have to write this yourself since you need to parse tokens that might be quoted with single or double quotes and may therefore contain whitespace or pipe characters within the quoted string. You also have to account for the pipe character possibly not having any whitespace before or after surrounding tokens. Using the *strtok* library will not work. You may assume a maximum of 50 arguments.

One possible way to construct the argument list for a command (but you do not necessarily have to follow this):

```
//define a 2-dimensional character array. Of course using a one-dimensional array could be more
//space efficient
char buf[50][100];

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

//step 1: copy each token into buf[i], e.g., using strcpy

//step 2: construct the argument list, assuming n arguments including the command
for (int i  = 0; i < n; i++)
      args[i] = (char*)buf[i];
args[n] = (char*)NULL;  //very important to end the argument list!!!!
```

Keep your functions small

Advice: resist the temptation to put a lot of logic into any one function. Keep the purpose of each function highly focused. For example, create separate functions to read lines, parse lines, and fork & exec.

Running commands

To run a command, use *fork()* and *execvp()*.

The child does an *execvp(argv[0], argv)*. This is a library function over *execve* that saves us the trouble of searching through the PATH environment variable.

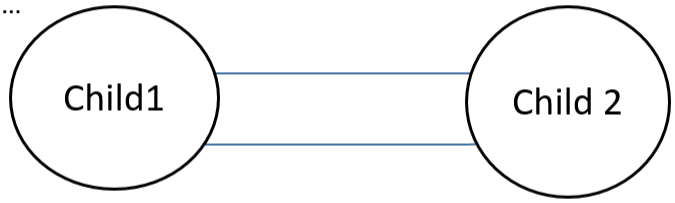
Handling pipe and dup2

Below is an example for a parent process to set up a pipe between its two children Child1 and Child2 for one-way data flow. Child1 writes its output data to standard output and Child2 reads its input data from standard input. Now with the pipe established, Child1's output data become Child2's input data.

In this project, you may have more than two commands. So you may have to manage multiple pipes (which can be created by the parent in advance. Why?) and assign them to different pairs of child processes.

Parent process:

```
int   fd[2];
pipe(fd); // make a pipe used for Child 1 sending data to Child2
...
```



Child 1:

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

Child 2:

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

Close all unused file descriptors

When a process forks a child process, the child process has a copy of all the open files in its parent process. Therefore, it is very important to close all unused file descriptors (including the file descriptors of the pipe) in each process including parent and child processes.
It is crucial that you close the file descriptors of the pipe in the parent. Otherwise, any processes reading from pipes will never exit.

Testing

In multi-processing programming, bugs seem to be complicated and daunting. Often, your process will hang. If it happens, please do not panic. The first thing to do is kill the process. Assume the process is "proj".

1. 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 proj
31055 pts/2    00:00:00 proj
31056 pts/2    00:00:00 ps
```

2. Use kill to kill all hanging processes (kill -9 pid)
   $ kill -9 31052
   $ kill -9 31055

You will see something like this:

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

Below I provide some test cases. You always can text your shell against the shell provided by the system to see if your shell behaves correctly.

Simple single command

- $ echo hello, world
  You should see some like:

  ```
  hello, world
  process 16589 exits with 0
  ```
- $ echo "abc" "def" 'ghi'   'jkl'
  You should see:

  ```
  abc def ghi jkl
  process 9649 exits with 0
  ```
- $ ls –l
  You should see something like:

  ```
  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
  ```

A command works with a single pipe

- $ ls -laF / | tr a-z A-Z
  You should see something 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

A command works with three pipes

- $ ls -alF / | grep bin | cat –n
  You should see something like this:
  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

A command works with more than three pipes

- $ ls -alF / | grep bin | tr a-z 'A-Z' |rev|cat –n
  You should see something like this:
  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

Pipes that pass a lot of data should work (If you shell can do this, applause!)

- $ cat proj1.cpp |tr A-Z a-z|tr -C a-z '\n' |sed '/^$/d' |sort|uniq -c|sort -nr|sed 10q
  You should see something like this:
  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
```

Submission

Before submission, you must make sure that your code is working on Linux server cs1 and your submission include all required files! You must run the submission command on cs1. You must make sure that your Makefile works properly (I assume that you have learned Makefile in the previous programming courses)!

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 p1.tar including the required files as specified above, by running the command:

```
tar -cvf p1.tar Readme *.h *.c Makefile
```

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

```
/home/fac/zhuy/class/SubmitHW/submit3500 p1 p1.tar
```

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

```
======================Copyright(C)Yingwu Zhu==========================
Wed Jan 17 21:53:58 PST 2018
Welcome testzhuy!
You are submitting array.cpp for assignment p1.
Transferring file.....
Congrats! You have successfully submitted your assignment! 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 shut down automatically once the deadline passes. You need to contact me for instructions on your late submission. Do not email me your submission!

Grading Criteria

Label	Notes
1a. Compilability & Complete submission (1 pt)	--- Your Makefile works properly. --- All required files are included in your submission.
1b. Readme, and Coding Format & Style (1 pt)	Readme is a text file, including: • Strengths • Weakness (any problems) • Team member names and their respective contributions, if it is a team project Coding format & style: • Clean, well-commented code.
1c. Functionality (6 pts)	The client and server programs behave as specified. 1. Working with one command 2. Working with one pipe 3. Working with two pipes 4. Working with more than two pipes 5. Working with multiple pipes & a lot of data
1d. Resource management, Outputs (2 pts)	1. Check return values for system calls. 2. No memory leaks (if applicable). 3. Each process closes unused file descriptors 4. Output: • Display messages as required (e.g., For each process that terminates, print that child's PID & exit status) • No debugging/testing messages • No messy messages scrambling on screen
1e. Overriding policy	If the code cannot be compiled or executed (segmentation faults, for instance), it results in zero point. If the submission is incomplete (e.g., missing files), it results in zero point.
1f. Late submission	Please refer to the late submission policy on Syllabus.

Q & A

Q1: How do I create pipes for n piped commands and assign them to the processes?
A1:
(1) You need to create n child processes P0, ..., Pn-1 for n commands 0, ..., n-1
(2) You need n-1 pipes for n processes:

```
int fd[n][2];
for (int i = 0; i < n-1; i++)
```

```
pipe(fd[i]);          //create n-1 pipes
```

This must be done in parent process before forking any child! Why???

(3) Assign pipes to processes:

You can assign them to processes in a manner such that process P_i (not the first process) read from pipe fd[i-1][0] and (not the last process) write to pipe fd[i][1].

In order to do so, you have to use dup2() to redirect I/O as specified in the project. This must be done before execvp()! Why???

(4) Use the same assignment relationship in (3) to close unused file descriptors in each child process before execvp()! Why???

Use a loop and if/else to get this done.

Q2: What will the main() in the program looks like?

A2: One possible way is shown as follows. Note that this is for illustration purpose, and you need always check the return value of system calls!

```
int main() {
      display the shell prompt and wait for user's command line;
      Parse the command line and store the command(s) into buffer(s);
      Let n be # of commands, then we need n-1 pipes
      int fd[n][2];
      for (int i = 0; i < n-1; i++)
         pipe(fd[i]);           //create n-1 pipes
      for (int i = 0; i < n; i++) {
         int ret = fork();
         if (ret == -1) {
            error handling;
            break;
          }
         if (!ret) { //child process
            close all not used file descriptors!!!
            construct arguments for execvp();
            call execvp(); //if you use dynamic memory, you never get a
                           // chance to release the memory!!!
            error handling…
         } else { //parent process
            …
         }
     }
     Parent process close all file descriptors
     Wait for all child processes using wait() and display each child process'
     process id and exit status;
     …
}
```

Q3: How to use execvp() in this project?

A3: Assume a command: echo "abc" "def"

```
char buf[50][100];
strcpy(buf[0], "echo");
strcpy(buf[1], "abc");
```

```
strcpy(buf[2], “def”);
char* args[50];
for (int i = 0; i < 3; i++)
    args[i] = (char*)buf[i];
args[3] = (char*)NULL;
execvp(args[0], args);
```

The above code is just an example for using execvp(). Your project code could do this differently depending on how you store the parsed command arguments in buffers.

<u>Appendix A:</u> Create your own Makefile

I assume that you used Makefile in prior classes. In case that you did not, here we provide an example of Makefile (**Note: the space preceding $(CC) must be tab!!!**)

```
#Makefile: if you use gcc, then change g++ to gcc
OBJS = MyShell.o
CC = g++
DEBUG = -g
CFLAGS = -Wall -c $(DEBUG)
LFLAGS = -Wall $(DEBUG)

myshell : $(OBJS)
    $(CC)  $(LFLAGS)  $(OBJS)  –o  myshell

MyShell.o : MyShell.h
    $(CC)  $(CFLAGS)  MyShell.cpp


clean:
    \rm *.o *~
```

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