See "Project 1 - UNIX Shell" in OSC, 10th edition, chapter 3, page P-12

What to submit? A single documented C file, **shell.c**

**Project 1—UNIX Shell**

* This project consists of designing a C program to serve as a **shell interface** that accepts **user commands** and then **executes** **each command** in a **separate process**.
* Your implementation will support **input** and **output** **redirection**, as well as **pipes** as a form of **IPC between a pair of commands**.
* Completing this project will involve using the UNIX fork(), exec(), wait(), dup2(), and pipe() system calls and can be completed on any Linux, UNIX, or macOS system.

**Overview**

* A shell interface gives the user a prompt, after which the next command is entered.
* The example below illustrates the prompt osh> and the user's next command: cat prog.c.  (This command displays the file prog.c on the terminal using the UNIX cat command.)

osh>cat prog.c

* One technique for implementing a shell interface is to have the **parent process first read what the user enters on the command line** (in this case, cat prog.c) and then create a separate child process that performs the command.
* Unless otherwise specified, the parent process waits for the child to exit before continuing.
* This is similar in functionality to the new process creation illustrated in Figure 3.9 in the book.
* However, UNIX shells typically also allow the child process to run in the background, or concurrently.  To accomplish this, we add an ampersand (&) at the end of the command.  Thus, if we rewrite the above command as

osh>cat prog.c &

the parent and child processes will run concurrently.

* The separate child process is created using the fork() system call, and the user's command is executed using one of the system calls in the exec() family (as described in Section 3.3.1 of the book).
* A C program that provides the general operations of a command-line shell is supplied in Figure 3.32 of the book also show below. The main() function presents the prompt osh-> and outlines the steps to be taken after input from the user has been read.
* The main() function continually loops as long as should\_run equals 1; when the user enters exit at the prompt, your program will set should run to 0 and terminate.

#include <stdio.h>

#include <unistd.h>

#define MAX\_LINE 80 /\* The maximum length command \*/

int main(void)

{

char \*args[MAX\_LINE/2 + 1]; /\* command line arguments \*/

int should\_run = 1; /\* flag to determine when to exit program \*/

while (should\_run) {

printf("osh>");

fflush(stdout);

/\*\*

\* After reading user input, the steps are:

\* (1) fork a child process using fork()

\* (2) the child process will invoke execvp()

\* (3) parent will invoke wait() unless command included &

\*/

}

return 0;

}

Figure 3.32 (from the book) Outline of simple shell.

This project is organized into several parts:

* **1.** Creating the child process and executing the command in the child
* **2.** Providing a history feature
* **3.** Adding support of input and output redirection
* **4.** Allowing the parent and child processes to communicate via a pipe

**Executing Command in a Child Process**

* The first task is to modify the main() function in Figure 3.32 so that a child process is forked and executes the command specified by the user.
* This will require parsing what the user has entered into separate tokens and storing the tokens in an array of character strings (args in Figure 3.32).
* For example, if the user enters the command ps -ael at the osh> prompt, the values stored in the args array are:

args[0] = "ps"  
  
args[1] = "-ael"  
  
args[2] = NULL

* This args array will be passed to the execvp() function, which has the following prototype:

execvp(char \*command, char \*params[])

* Here, command represents the command to be performed and params stores the parameters to this command.
* For this project, the execvp() function should be invoked as execvp(args[0], args).
* Be sure to check whether the user included & to determine whether or not the parent process is to wait for the child to exit.

**Redirecting Input and Output**

* Your shell should then be modified to support the ‘>’ and ‘<’ redirection operators, where ‘>’ redirects the output of a command to a file and ‘<’ redirects the input to a command from a file.
* For example, if a user enters

osh>ls > out.txt

the output from the ls command will be redirected to the file out.txt.

* Similarly, input can be redirected as well.
* For example, if the user enters

osh>sort < in.txt

the file in.txt will serve as input to the sort command.

* Managing the redirection of both input and output will involve using the dup2() function, which duplicates an existing file descriptor to another file descriptor.
* For example, if fd is a file descriptor to the file out.txt, the call

dup2(fd, STDOUT\_FILENO);

duplicates fd to standard output (the terminal).  This means that any writes to standard output will in fact be sent to the out.txt file.

* You can assume that commands will contain either one input or one output redirection and will not contain both.  In other words, you do not have to be concerned with command sequences such as sort < in.txt > out.txt.