CPSC 351: Unix Shell Project - Fall 2023

Unix Shell Project, due by Sunday, 17 Sep 2023

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 on Linux, UNIX, or macOS.

I. 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.

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 exec() system calls.
A C program that provides the general operations of a command-line shell is supplied in Figure 3.36. 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 <string.h>
#include <stdlib.h>
#include <stdbool.h>
#include <sys/types.h>
#include <unistd.h>

#define MAX_CMD_LINE_ARGS 128

int min(int a, int b) { return a < b ? a : b; }

    // break a string into its tokens, putting a \0 between each token
    // save the beginning of each string in a string of char *'s (ptrs to chars)
int parse(char* s, char* argv[]) {
    const char break_chars[] = " \t;";
    char* p;
    int c = 0;
          // TODO
    return c; // int argc
}


    // execute a single shell command, with its command line arguments
    // the shell command should start with the name of the command
int execute(char* input) {
    int i = 0;
    char* shell_argv[MAX_CMD_LINE_ARGS];
    memset(shell_argv, 0, MAX_CMD_LINE_ARGS * sizeof(char));

    int shell_argc = parse(input, shell_argv);
    int status = 0;
    pid_t pid = fork();

    if (pid < 0) {
        fprintf(stderr, "Fork() failed\n");          // send to stderr
    } else if (pid == 0) {                           // child
        int ret = 0;
        if ((ret = execvp(*shell_argv, shell_argv)) < 0) {
            fprintf(stderr, "execvp failed\n");
        }
            printf("\n");
    }
    else {                                           // parent
        while (wait(&status) != pid) { }
    }
    return 0;
}

int main(int argc, const char * argv[]) {
    char input[BUFSIZ];
    char last_input[BUFSIZ];
    bool finished = false;

    memset(last_input, 0, BUFSIZ * sizeof(char)); // clear memory
    while (!finished) {
```

```
        memset(input, 0, BUFSIZ * sizeof(char));
        printf("osh > ");
        fflush(stdout);

        if (strlen(input) > 0) {
            strncpy(last_input, input, min(strlen(input), BUFSIZ));
            memset(last_input, 0, BUFSIZ * sizeof(char));
        }

        if ((fgets(input, BUFSIZ, stdin)) == NULL) {       // or gets(input, BUFSIZ);
            fprintf(stderr, "no command entered\n");
            exit(1);
        }
        input[strlen(input) - 1] = '\0'; // wipe out newline at end of string
        if (strncmp(input, "exit", 4) == 0) {              // only compare first 4 letters
            finished = true;
        } else if (strncmp(input, "!!", 2) == 0) {             // check for history command
                  // TODO
        } else {
            execute(input); }
        }
    printf("\t\t...exiting\n");
    return 0;
}


    /**
    * (1) read user input
    *
    * (2) parse the input (a bunch of pointers to strings
    *     into a single string separated by '\0', with a separate array
    *     of pointers (char*), pointing to the start of each mini string
    *     e.g., 'one\0two\0three\0four\0' (afer processing)
    *     [p1 p2 p3 p4] <-- char* shell_argv
    *     4 <-- int shell_argv
    *
    * (3) if shell_argv[0] is "exit", then
    *     finished = true; continue; // (go to top of while) OR
    *     exit(0); // just exit
    *
    * (3) fork a child process
    *
    * (4) child process will invoke execvp() using its processed
    *     shell_argv variables
    *
    * (5) parent will invoke wait() (unless command included &
    *     to indicate the child should be a daemon process
    */
    }
     return 0;
}
```

Figure 3.36 Outline of simple shell.

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**
5. **This project is organized into several parts:**

II. Executing Command in a Child Process

The first task is to modify the main() function in Figure 3.36 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 3.36). 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.

III. Creating a History Feature

The next task is to modify the shell interface program so that it provides a history feature to allow a user to execute the most recent command by entering !!.

For example, if a user enters the command

ls -l,

she can then execute that command again by entering !! at the prompt. Any command executed in this fashion should be echoed on the user's screen, and the command should also be placed in the history buffer as the next command.

Your program should also manage basic error handling. If there is no recent command in the history, entering!!should result in a message "No commands in history."

IV. 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.
```

.

V. Communication via a Pipe

The final modification to your shell is to allow the output of one command to serve as input to another using a pipe.

For example, the following command sequence

```
osh > ls -l | less
```

VI. Submission

Submit to (a) Canvas all .c and .h project files, example output of your program as a comment at the bottom of the shell program, and a gif file showing the program being executed, AND (b) **the completed rubric below.** If you don't fill out the rubric you will lose 10% of your grade.

CPSC 351 project --- Unix Shell **Verify each of the following items and place a checkmark in the correct column. Entries incorrectly marked will incur a 5% penalty on the grade for this assignment**	**Due 17 Sep 2023**	
Name(s) and section: (do not leave the checkboxes unmarked)	Completed	Not Completed
Create a shell program that displays the prompt "osh >" each time it is waiting for a command	☐	☐
The shell program parses the user command by placing 0's into the spaces in between words, and returns an array of char* pointers to the words in the command (const char* argv[])	☐	☐
The shell program checks to see if "exit" has been entered, and exits the shell if it has	☐	☐
The shell program repeats the last command if "!!" has been entered	☐	☐
The shell program executes commands with no arguments using fork/exec (e.g., ls, or pwd)	☐	☐
The shell program executes commands with 1 argument using fork/exec (e.g., ls -la, or cal 2023)	☐	☐
The shell program executes commands with 2 or more arguments using fork/exec and an array of arguments ending with NULL	☐	☐
The shell program executes commands that can take input from a file and send output to a file (e.g., cat < in.txt > out.txt), using dup2	☐	☐
The shell program can use a pipe to send output from one program to another (e.g., ls -l | out.txt)	☐	☐
Has output showing the program running as a comment in the bottom of the shell course	☐	☐
Created a working gif file showing the shell program executing its functions	☐	☐
Additional comments…		