Jacob Gaylord

COSC 310 Operating Systems

22 May 2019

Writing a Shell in C

Abstract

The purpose of this research project is to build a feature-rich shell, similar to a bash shell. Creating a shell in C familiarizes the developer with how a shell works and teaches C programming skills. This shell translates ASCII input into system calls and displays the output to the user. The first attempt to develop this project was aimed at creating a shell with useful functions not included in the bash shell. This idea became a roadblock because of my limited knowledge of C. The new aim of this shell is to create a simple, yet useful shell with easy to understand source code. To begin, the introduction goes over the basics of a bash shell and shell implementation. The methods portion contains the C libraries used and how each function operates. The descriptions included in this document thoroughly explain each primary feature in the source code and are accompanied by test output. Finally, the results and future additions are discussed.

Introduction

The shell is not part of the physical operating system, but a program that interprets user commands into system calls for the OS to execute. At its most basic, a shell is a simple loop that prints a prompt, reads user commands, and runs the requested system call or built-in function. Built-in functions can change properties of the parent process (the shell) whereas

Methods

Each function used in the shell comes from the following libraries, stdbool, stdlib, stdio, string, sys/wait, unistd. Each library defines functions that are used in the shell implementation.

stdbool

This library defines bool macros to improve readability and reduce programming mistakes. The bool type is smaller than the int type and therefore use less memory.

stdlib

The stdlib defines the following functions and macros that are used in the shell, exit, EXIT_SUCCESS, EXIT_FAILURE, free, malloc, and realloc. The exit function terminates a program typically using the EXIT_SUCCESS and EXIT_FAILURE macros to signal how the program ended. These macros are used mostly for readability. The library functions free, malloc, and realloc are all memory allocation tools. The malloc function manually allocates memory, realloc reallocates memory, and free deallocates memory previously allocated by malloc or realloc.

stdio

The I/O library used is stdio, which supplies the functions for printf, perror, and stderr. The function printf is used for printing non-error messages to the screen. If an error message is to be printed, then perror is used. Perror does not write to stdout but rather the stderr output stream. The stderr output stream is applied over the stdout stream to separate error messages from standard messages (helpful when implementing redirection).

string

The string library contains several useful functions for string manipulation. To parse a multiple word string into different tokens, strtok was used. This is done by passing a char pointer and a set of delimiters to split your string by. The other function used is strcmp to check to see if a task is built-in.

sys/wait

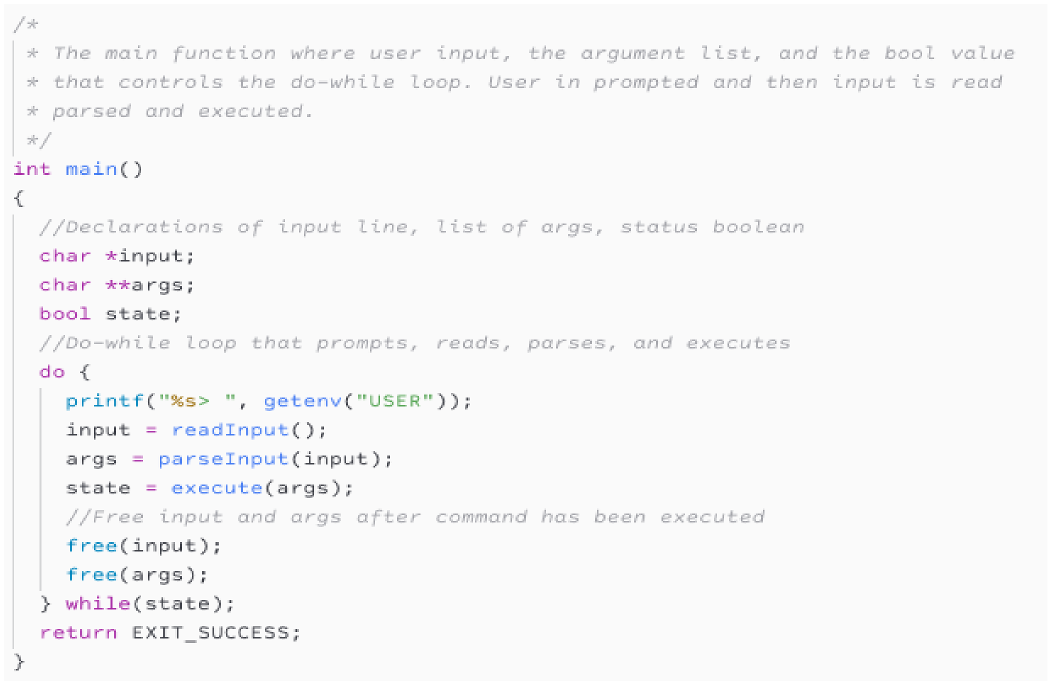
The sys/wait header defines several macros useful for waiting for a child process to complete. The first macro is WUNTRACED, which reports the status of the child process and is passed to the waitpid function. The macro WIFEXITED returns true if the child process exited normally and false if an error occurred. The last macro used is WIFSIGNALED, which returns true if a child exited due to an uncaught signal and false if not.

unistd

The unistd library defines useful functions like chdir, execvp, and fork. The chdir function is used in the built-in cd function to change to the directory of the current process image. The execvp function is used to start any process that is not built-in. Execvp takes a file, and any arguments passed to the program, then replaces the current process image with the program specified. The fork function is used in the launch function to split up processes into children to protect the parent shell process from being exited prematurely.

Results

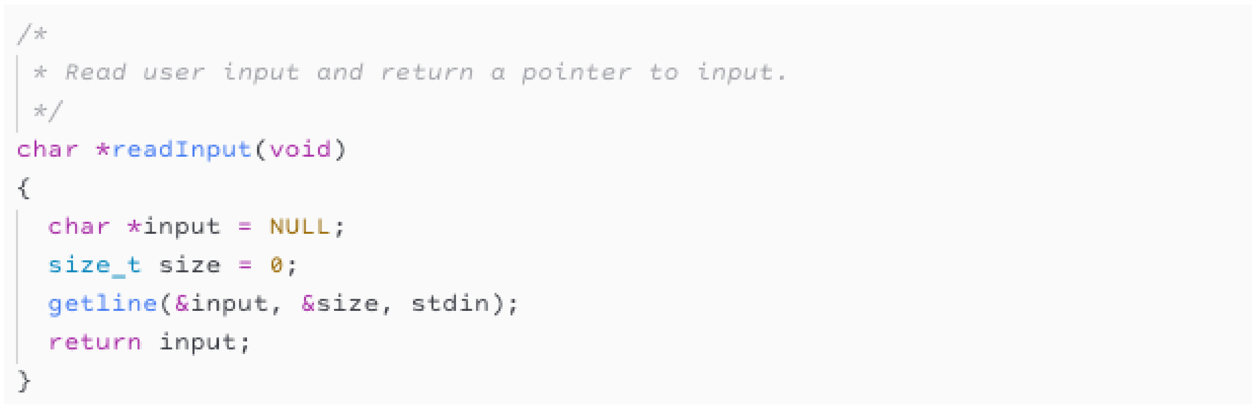
The first step in creating this shell was getting a basic framework, and then defining functions from there. The basic life of a shell reads parses and executes in a loop until the user decides to exit. After every line of output, the arguments are then freed, and the loop starts over.



```c
/*
 * The main function where user input, the argument list, and the bool value
 * that controls the do-while loop. User in prompted and then input is read
 * parsed and executed.
 */
int main()
{
  //Declarations of input line, list of args, status boolean
  char *input;
  char **args;
  bool state;
  //Do-while loop that prompts, reads, parses, and executes
  do {
    printf("%s> ", getenv("USER"));
    input = readInput();
    args = parseInput(input);
    state = execute(args);
    //Free input and args after command has been executed
    free(input);
    free(args);
  } while(state);
  return EXIT_SUCCESS;
}
```

The main function starts by defining the underlying data structure utilized by the program. These declarations include the input, arguments to be passed, and a bool variable to keep exit the program if the user wishes to exit. Next, a do-while loop is used so that the first time the program is run, the state does not need to be checked. The user environment variable is printed as a prompt. Then the input is read parsed and executed by their three respective functions readInput, parseInput, and execute. Once the input is read in, it is stored in input. After the input string has been parsed, a list of strings is stored in args. These arguments are then passed to the execute function, which returns a bool stored in state. If the main shell program exits with no errors, then EXIT_SUCCESS is returned.

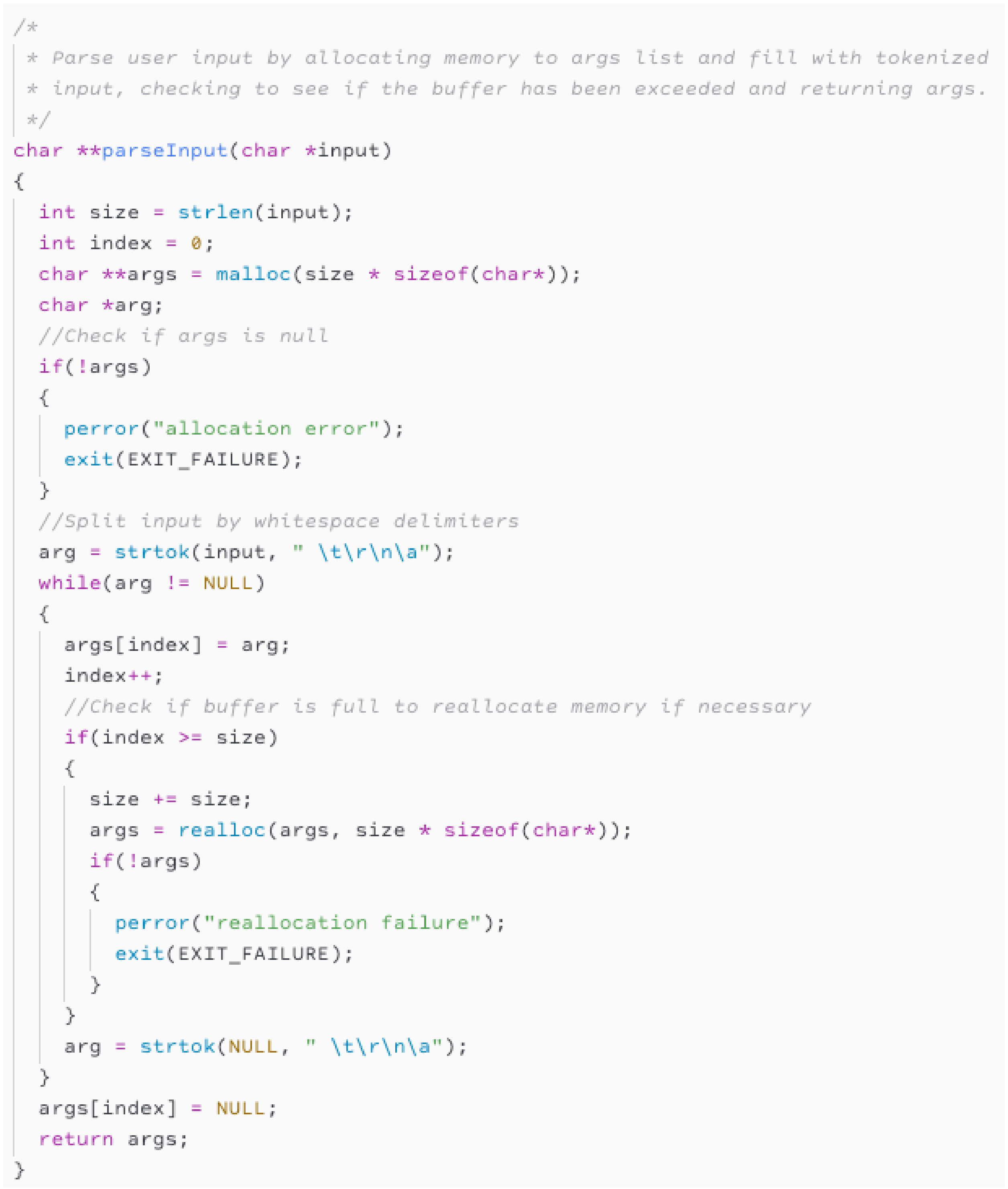
When reading input, C makes it very easy to dynamically allocate memory to the size of the input through the getline function.



```c
/*
 * Read user input and return a pointer to input.
 */
char *readInput(void)
{
    char *input = NULL;
    size_t size = 0;
    getline(&input, &size, stdin);
    return input;
}
```

A pointer to a char is initialized to NULL because getline requires that the memory be freeable. A size_t variable is declared and is given the value of 0 to initialize a variable address that contains the size of the input buffer. The getline function then dynamically allocates a buffer based on the size of the string, and the input string is returned to the caller.

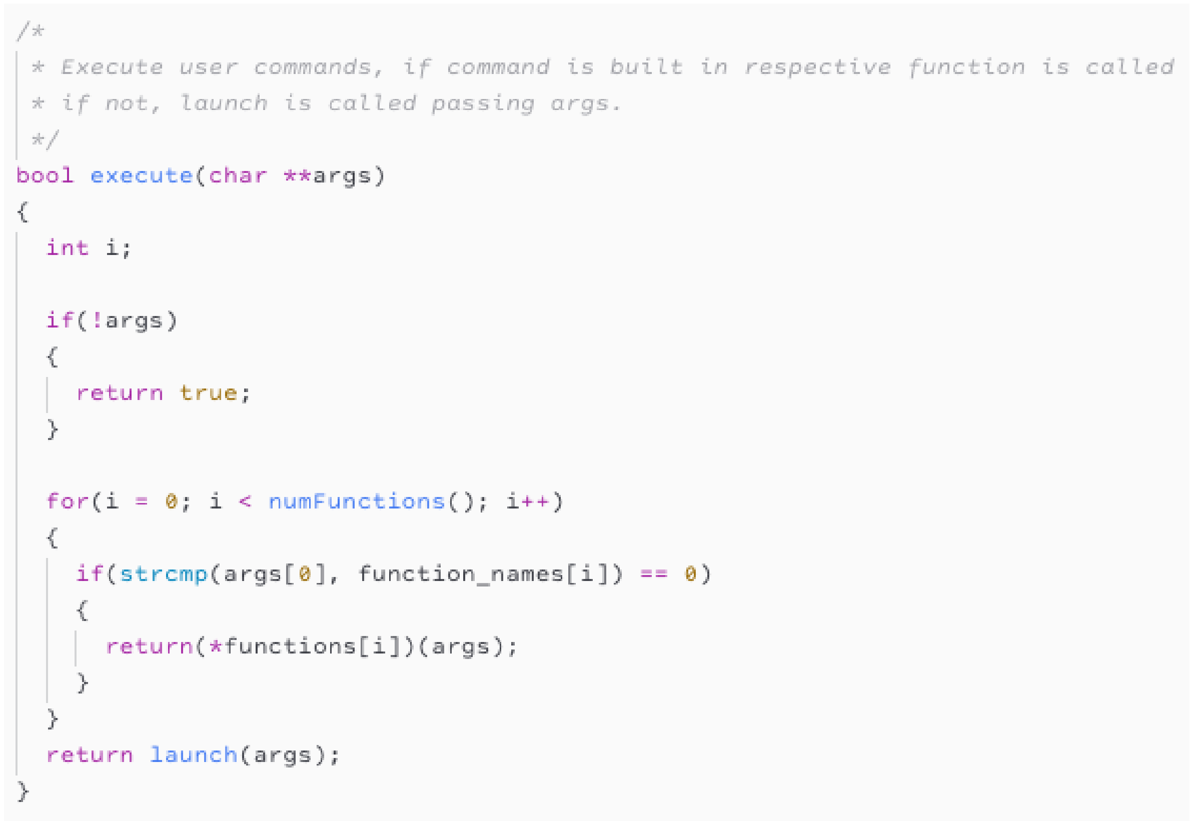
The next step of the process is to parse the input string into a list of arguments to be utilized by a built-in function or execvp. Unlike reading input, parsing input requires the programmer to dynamically allocate a buffer based on the length of the arguments passed.



```c
/*
 * Parse user input by allocating memory to args list and fill with tokenized
 * input, checking to see if the buffer has been exceeded and returning args.
 */
char **parseInput(char *input)
{
  int size = strlen(input);
  int index = 0;
  char **args = malloc(size * sizeof(char*));
  char *arg;
  //Check if args is null
  if(!args)
  {
    perror("allocation error");
    exit(EXIT_FAILURE);
  }
  //Split input by whitespace delimiters
  arg = strtok(input, " \t\r\n\a");
  while(arg != NULL)
  {
    args[index] = arg;
    index++;
    //Check if buffer is full to reallocate memory if necessary
    if(index >= size)
    {
      size += size;
      args = realloc(args, size * sizeof(char*));
      if(!args)
      {
        perror("reallocation failure");
        exit(EXIT_FAILURE);
      }
    }
    arg = strtok(NULL, " \t\r\n\a");
  }
  args[index] = NULL;
  return args;
}
```

In parseInput, four variables are declared for use within the function. The size variable holds the length of the input string passed to the function. The index variable is used to check if the current buffer has been exceeded. Then memory is manually allocated to a variable args by multiplying the string length and the size of a char*. In order to store individual arguments and arg variable is declared. The first check determines if args is empty if it is then an error message is printed and exited passing exit failure. This would be the result of an allocation error, meaning that there wasn't enough memory available for use to allocate a large enough buffer. Next, the function begins to tokenize the input. Next, a while loop is used to iterate over the args and begins by assigning the first token to the beginning of the list. The index variable is incremented, and the buffer size is checked here. If the buffer is full, the size variable and memory allocated to args is doubled. If there is an error reallocating memory, then an error message is printed, and the program is exited. Lastly, the next argument is tokenized, and the loop continues. If the end of the arguments has been reached, they are returned to the caller.

The arguments are then passed to the execute function which calls the respective built-in function or launches a new process.



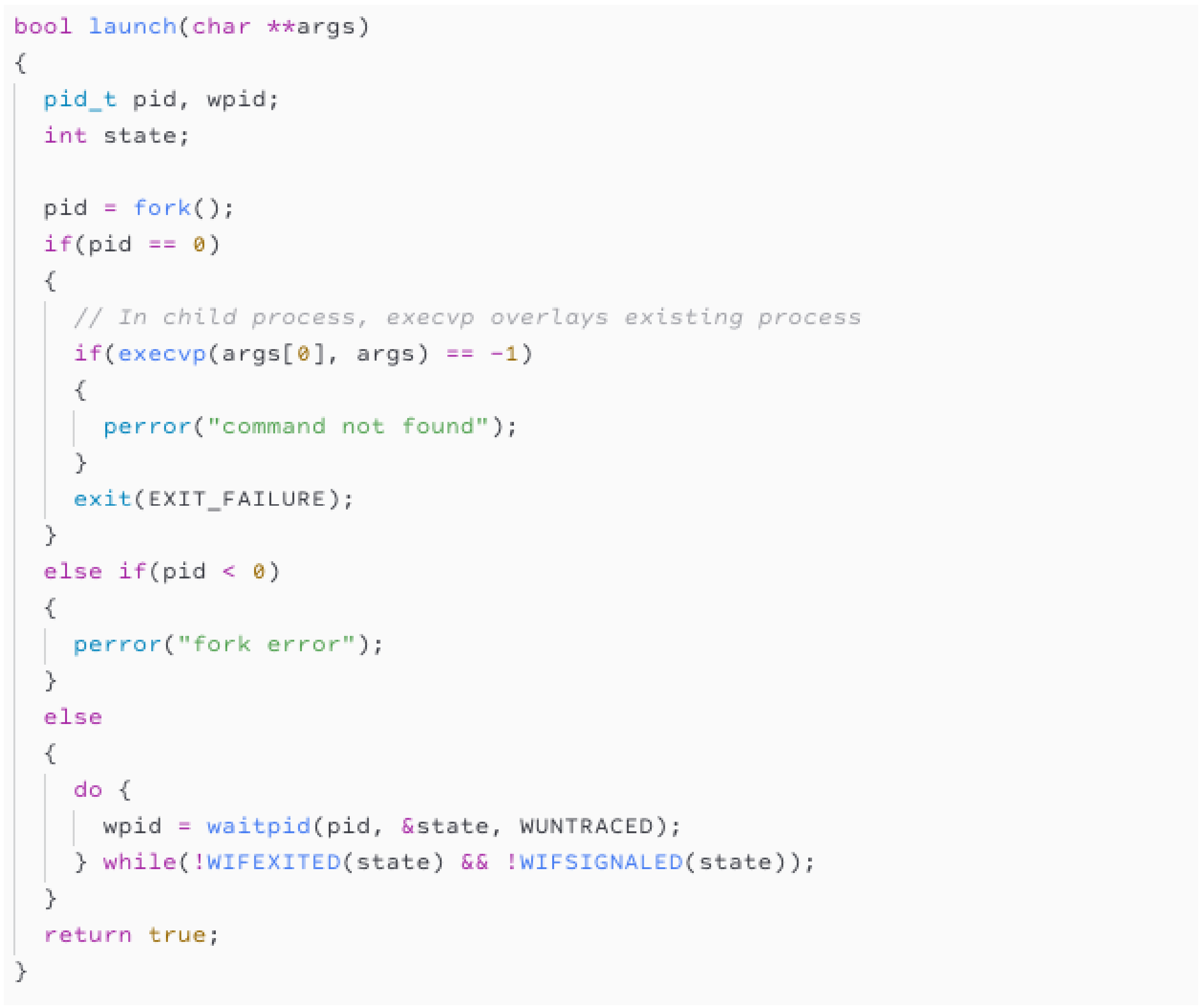
```c
/*
 * Execute user commands, if command is built in respective function is called
 * if not, launch is called passing args.
 */
bool execute(char **args)
{
  int i;

  if(!args)
  {
    return true;
  }

  for(i = 0; i < numFunctions(); i++)
  {
    if(strcmp(args[0], function_names[i]) == 0)
    {
      return(*functions[i])(args);
    }
  }
  return launch(args);
}
```

First, if the args pointer is null, then a true bool value is passed to the caller to cancel execution and reprompt. To check if the user entered a built-in function the list of function names is compared to the first argument for the number of functions defined if a function name matches then a pointer to its respective function passing it args is returned to the caller. If no function is found then launch is run passing args and returned to the caller.

To start a new process, the launch function is called. This function creates a new process and executes the respective command while the parent process waits for completion.

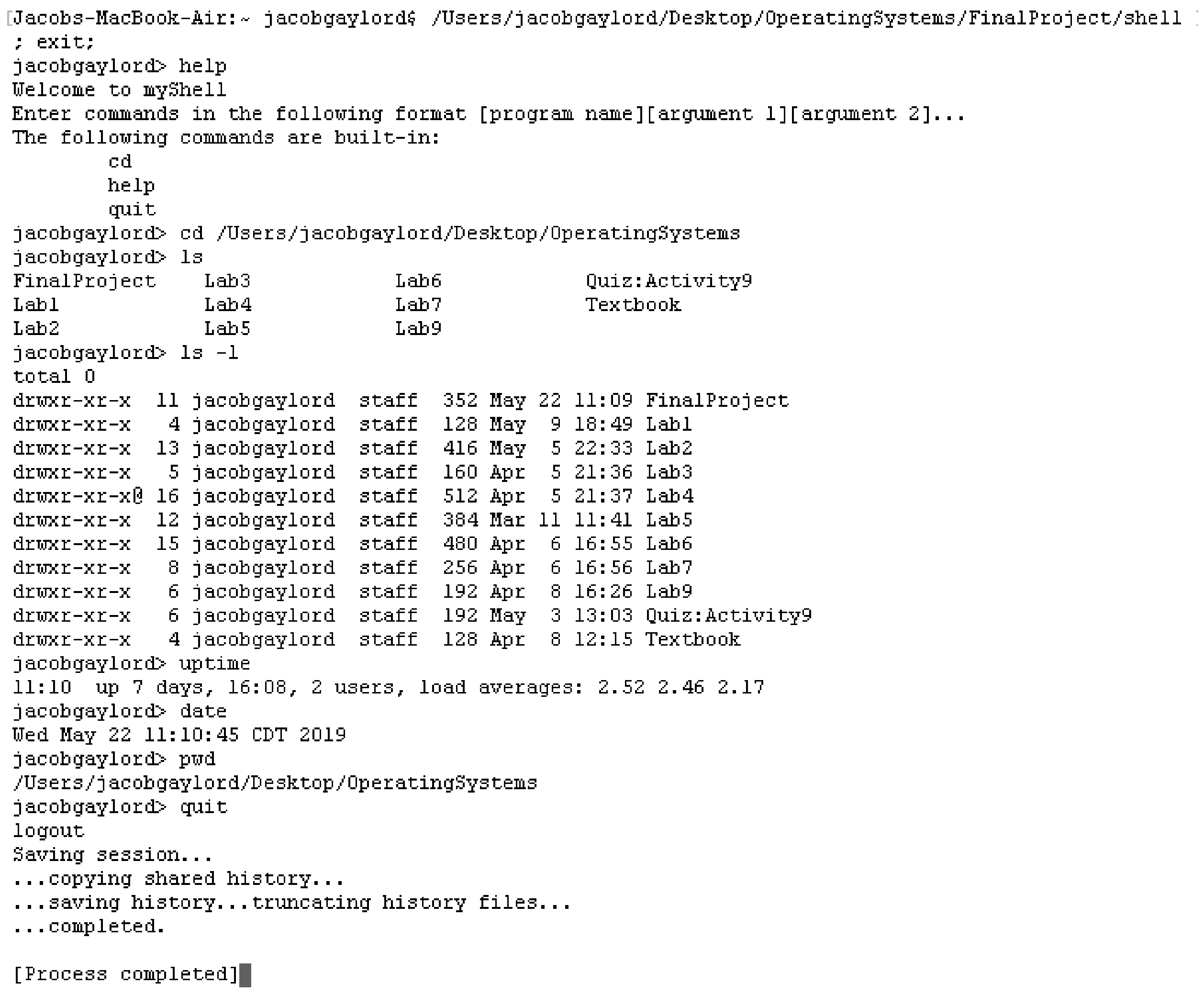


```
bool launch(char **args)
{
    pid_t pid, wpid;
    int state;

    pid = fork();
    if(pid == 0)
    {
        // In child process, execvp overlays existing process
        if(execvp(args[0], args) == -1)
        {
            perror("command not found");
        }
        exit(EXIT_FAILURE);
    }
    else if(pid < 0)
    {
        perror("fork error");
    }
    else
    {
        do {
            wpid = waitpid(pid, &state, WUNTRACED);
        } while(!WIFEXITED(state) && !WIFSIGNALED(state));
    }
    return true;
}
```

To begin, the function defines two process IDs for the waiting and child process. Next, a fork is called, and a new process is created. The execvp function is called passing the arguments and is checked for errors if an error is found the shell exits. If the pid is less than zero, than there was a fork error and an error message is printed. The waiting process and returns true when it has completed.

This basic shell can run most basic commands but does not support piping or redirection. It is capable of handling flags and can be used for very simple tasks. The following is a portion of test output run on a Unix machine to demonstrate basic functionality.



```
[Jacobs-MacBook-Air:~ jacobgaylord$ /Users/jacobgaylord/Desktop/OperatingSystems/FinalProject/shell
; exit;
jacobgaylord> help
Welcome to myShell
Enter commands in the following format [program name][argument 1][argument 2]...
The following commands are built-in:
        cd
        help
        quit
jacobgaylord> cd /Users/jacobgaylord/Desktop/OperatingSystems
jacobgaylord> ls
FinalProject    Lab3            Lab6            Quiz:Activity9
Lab1            Lab4            Lab7            Textbook
Lab2            Lab5            Lab9
jacobgaylord> ls -l
total 0
drwxr-xr-x  11 jacobgaylord  staff  352 May 22 11:09 FinalProject
drwxr-xr-x   4 jacobgaylord  staff  128 May  9 18:49 Lab1
drwxr-xr-x  13 jacobgaylord  staff  416 May  5 22:33 Lab2
drwxr-xr-x   5 jacobgaylord  staff  160 Apr  5 21:36 Lab3
drwxr-xr-x@ 16 jacobgaylord  staff  512 Apr  5 21:37 Lab4
drwxr-xr-x  12 jacobgaylord  staff  384 Mar 11 11:41 Lab5
drwxr-xr-x  15 jacobgaylord  staff  480 Apr  6 16:55 Lab6
drwxr-xr-x   8 jacobgaylord  staff  256 Apr  6 16:56 Lab7
drwxr-xr-x   6 jacobgaylord  staff  192 Apr  8 16:26 Lab9
drwxr-xr-x   6 jacobgaylord  staff  192 May  3 13:03 Quiz:Activity9
drwxr-xr-x   4 jacobgaylord  staff  128 Apr  8 12:15 Textbook
jacobgaylord> uptime
11:10  up 7 days, 16:08, 2 users, load averages: 2.52 2.46 2.17
jacobgaylord> date
Wed May 22 11:10:45 CDT 2019
jacobgaylord> pwd
/Users/jacobgaylord/Desktop/OperatingSystems
jacobgaylord> quit
logout
Saving session...
...copying shared history...
...saving history...truncating history files...
...completed.

[Process completed]
```

Discussion and Future Additions

The original idea for this project was to develop an entire useful shell application in C. As I began to add functionality to my shell I quickly began to realize that I couldn't do what I wanted to in C. My future plans for this project include writing a new version in C++ that has useful functions and nice output. This project became research into the inner workings of a shell, and how processes are created and user commands are executed. I learned quite a bit from some tutorials I found online, which I have included in my references. The tutorials gave me a framework for my shell, but to understand how the code worked, I used the POSIX specifications and the Linux man pages which I have also cited below.

Resources

The POSIX Specifications. https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/

Linux Man Pages. https://linux.die.net/man/2/

Brennan, Stephen. "Tutorial - Write a Shell in C." *Brennan.io*, 16 Jan. 2015, brennan.io/2015/01/16/write-a-shell-in-c/.

Gupta, Indradhanush. "Writing a Unix Shell." *Indradhanush.github.io*, 28 May 2017, indradhanush.github.io/blog/writing-a-unix-shell-part-1/.