

Simple Shell (SiSH)

Nathaniel Vallen

32221415

Global SW Convergence

Free Days: 5

List of Contents

Introduction.....	2
Important Concepts & Considerations.....	4
Unique Features.....	5
Build Configurations & Environment.....	6
Use of AI-based Assistance.....	6
Screen Captures.....	7
Personal Opinion.....	10

Introduction

This project is a simple shell program that takes in commands in the form of a singular input string and executes it by making a system call. A system call can be understood as a request made by the user (by giving a command) to the operating system (OS) to execute a specific command and access hardware if necessary. This requesting process is crucial to prevent potential conflicts that may arise, such as hardware usage conflicts and process conflicts.

Essentially, a shell can be seen as a program that serves as a medium for a user to make system calls (requests) to the operating system. This is done by looking for the command (executable file) in the PATH environment variable. If the command is available and written in the proper syntax, then a system call will be made. Otherwise, the shell is going to print out an error message and the user is prompted to enter a new command. To do this, the simple shell program is set to keep receiving commands (inputs) until the command is detected as “quit”, upon which the program will terminate. If the command is not a terminating command (“quit”), then the program will proceed to tokenize the command and store each token in an array. After that, the program fetches the local PATH environment variable and saves it. Now that we have both the command and PATH, we can look for the command (the name of the executable file) in the PATH environment variable. However, PATH separates each directory with the colon symbol (:), hence we need to first tokenize PATH and look for the executable file name in each directory (token). If no said command is found, then the shell will return to its initial state and prompt the user to give another input (command). Otherwise, it will fork, or in other words, create a new (child) process which is the command the user has made as the input. In this stage, it is difficult for us to exactly determine which process is going to run first (between parent and child), therefore, we have to set some conditions for both scenarios. If the parent process runs first, then it must wait until the child process is terminated. If the child process runs first, then it will run the command (execute) and once it's done, it will send a signal (status) which will be received by the parent process and terminate itself. In summary, the process can be described as follows:

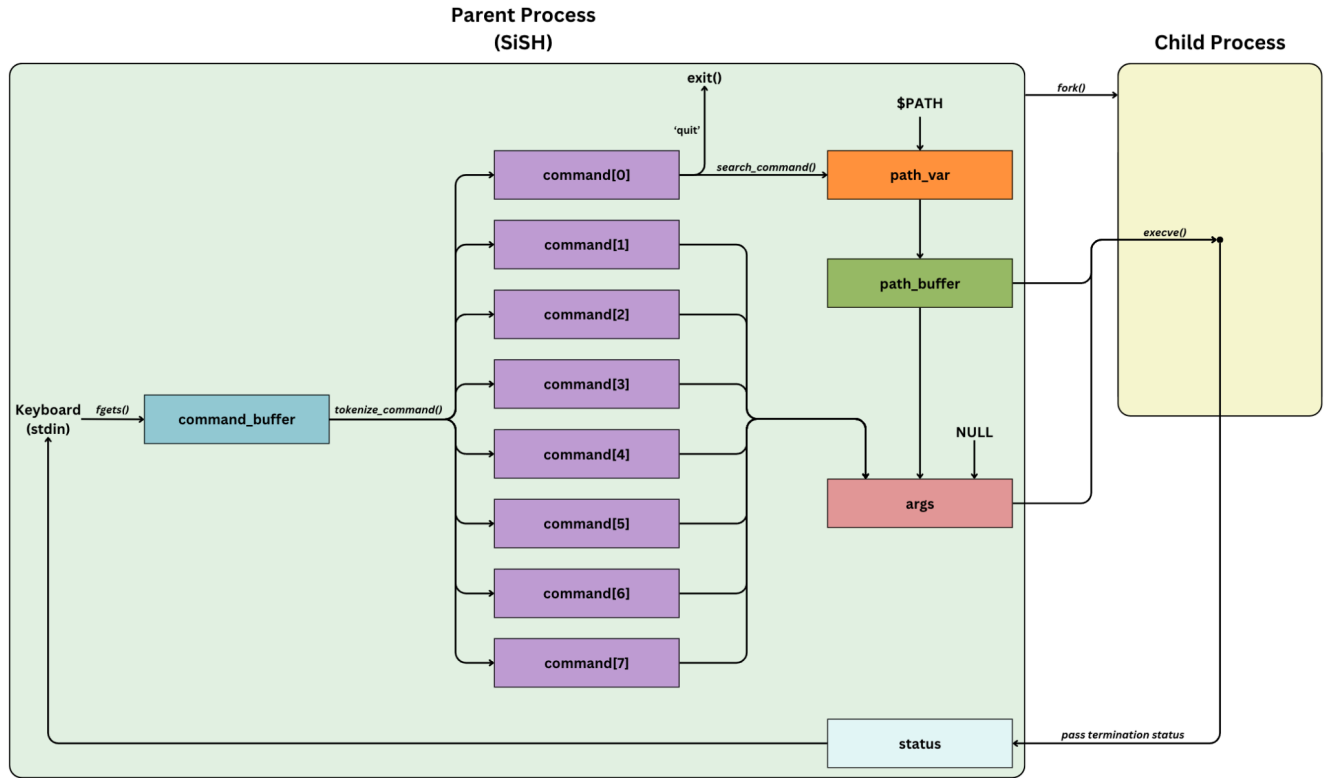


Figure 1. Process workflow

The search process can be seen in more details like the following:

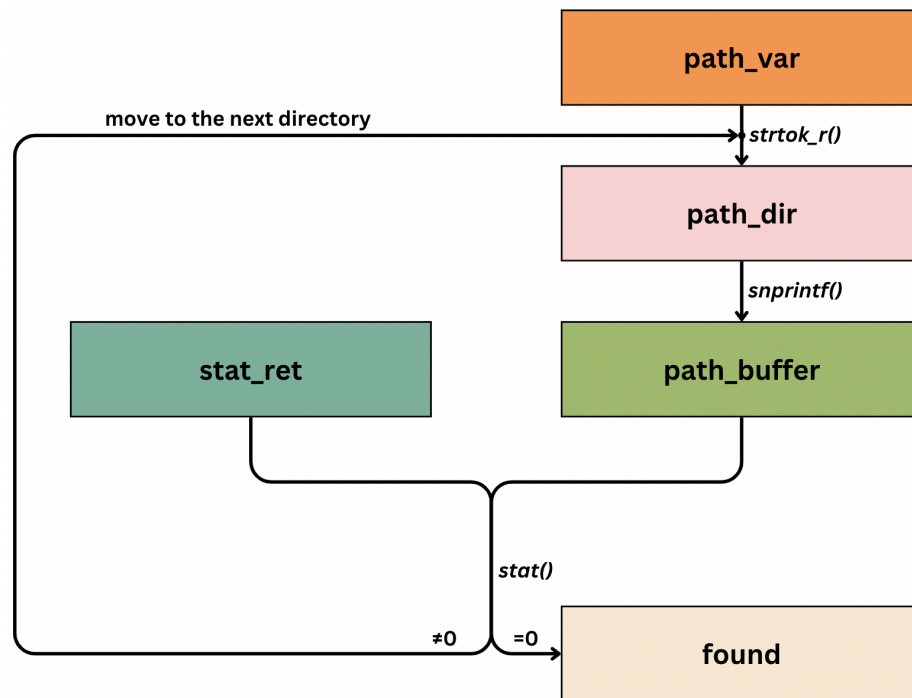


Figure 2. Workflow of search_command()

Important Concepts & Considerations

I. Below are the details of the variables and macros used in the program:

Name	Type	Description
args	char *[] (array of char pointers)	An array of arguments to be passed to <i>execve()</i>
checkpoint	char *	A variable that saves the tokenization state across calls
command	char *[] (array of char pointers)	A variable that contains tokens of a command
command_buffer	char []	A variable that contains the current command string
command_copy	char *	A copy of the command string to be tokenized
current_time	struct tm	A variable that holds the current time value
directory	char *	A variable that holds the current working directory
found	bool	An indicator if a command is found in PATH
loop_counter	int	A loop counter to help limit a loop and serve as an index
MAX_BUFF	macro	Maximum limit of the command buffer
MAX_CMD	macro	Maximum limit of a token of the command buffer
MAX_PATH	macro	Maximum limit of the path buffer
path_buffer	char []	A buffer to contain the path directory to be compared to the command (in <i>search_command()</i>)
path_copy	char *	A copy of path_var to be tokenized

path_dir	char *	A variable that holds the return value of tokenization (<i>in search_command()</i>)
path_var	char *	A variable that holds the PATH
pid	pid_t	A variable that holds the process ID (PID) of the current running process
status	int	A return value from a child process once it's terminated
stat_ret	struct stat	A variable that holds the related details of a file (as a return value of <i>stat()</i>)
token	char *	A variable that holds the return value of tokenization (<i>in tokenize_command()</i>)
tv	struct timeval	A structure that holds time intervals in seconds and microseconds
tz	struct timezone	A structure that holds the local time zone
user	char *	A variable that holds the username

- II. The program consists of two C files (main.c and functions.c) alongside a header file (functions.h) and a Makefile.
- III. Study references used in this project are mostly from class materials, Linux man pages, informational websites, and ChatGPT (this part specifically will be covered in a later section).

Unique Features

- A toggleable feature is available in the commented out section of the code. Once uncommented, the program will be able to print out additional information including:
 - The absolute path to the command
 - The arguments passed
- This project was complemented with a makefile (instructions such as “make” and “make clean” can be used for creation/compilation and deletion respectively).
 - By default, after the command “make” is entered, the executable file will be renamed to “app”.

- The “make” process automatically removes unused object files, leaving only the executable file.
- The default shell prompt has been customized to display more information including the username, the current working directory, and the current time (in the format of *hour:minute:second*).
- The shell is capable of taking in additional inputs (or flags) after a command.
- The shell has the ability to work with directory-related commands such as “cd” and “pwd”.

Build Configurations & Environment

The code in this project was written in C programming language using Visual Studio Code IDE.

Use of AI-based Assistance

During the working process of this project, I made some consultations to ChatGPT, mainly regarding the following items:

- The usage of *stat()* function in C

From this question, I gained the understanding of how the *stat()* function works in C. There was also some information about the *stat* structure included in the *sys/stat.h* header file. The *stat* structure holds a series of information related to a specific file which will be passed by the *stat()* function as return values. Additionally, ChatGPT recommended the implementation of the following three properties:

- *S_ISREG()* to check if the said file is a regular file.
- *S_IXUSR* to check if the current user has access to execute the file.
- *st_mode* contains the type and permissions of the file.

These are used in the codework to ensure that the target executable file is indeed executable in regards to file type and permission.

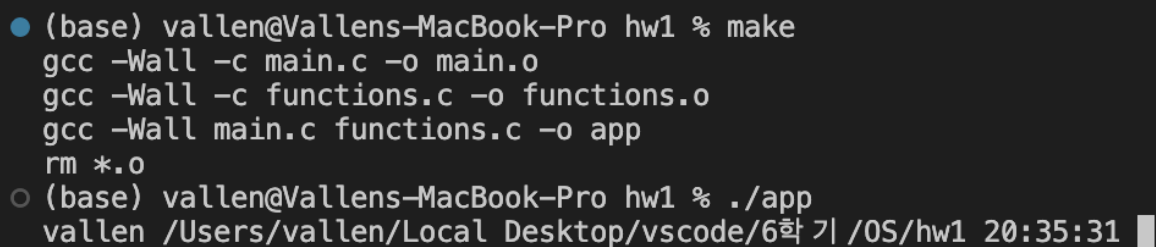
- The usage of *execve()* function in C

I asked ChatGPT about the precise usage and parameters of `execve()`. Although the purpose of this function was taught in class, I went to look deeper for the exact parameters used in this function, which can be divided into three parts:

- Pathname (Path Name)
 - The path to the designated executable file.
- Arguments
 - The list of arguments that follow the command.
- Environment
 - An array of strings that represent the environment variables for the new process.
 - In this case, it is set to NULL to inherit the parent process's environment variables.
- Environment variables and how they work

The questions I asked are mostly related to the environment variable PWD and its scope (how it is treated when a process is forked, etc.). From the answers, I have learned that the PWD environment variable is local to a process and when a process forks a child process, then the child process will also have the same PWD details as its parent process.

Screen Captures



```
● (base) vallen@Vallens-MacBook-Pro hw1 % make
gcc -Wall -c main.c -o main.o
gcc -Wall -c functions.c -o functions.o
gcc -Wall main.c functions.c -o app
rm *.o
○ (base) vallen@Vallens-MacBook-Pro hw1 % ./app
vallen /Users/vallen/Local Desktop/vscode/6학기/OS/hw1 20:35:31
```

Figure 3. Example of a “make” and “./app” output


```

vallen /Users/vallen/Local Desktop/vscode/6학기 /OS/hw1 20:53:41 ls

Path to command: /bin/ls
args is /bin/ls

Makefile      app      functions.c    functions.h    main.c      testfolder
vallen /Users/vallen/Local Desktop/vscode/6학기 /OS/hw1 20:53:43 ls -a -l -h

Path to command: /bin/ls
args is /bin/ls -a -l -h

total 120
drwxr-xr-x  10 vallen  staff   320B Sep 28 20:53 .
drwxr-xr-x   4 vallen  staff  128B Sep 14 22:56 ..
-rw-r--r--@  1 vallen  staff   6.0K Sep 27 13:20 .DS_Store
drwxr-xr-x   3 vallen  staff   96B Sep 24 10:56 .vscode
-rw-r--r--   1 vallen  staff  236B Sep 28 14:55 Makefile
-rwxr-xr-x   1 vallen  staff   34K Sep 28 20:53 app
-rw-r--r--   1 vallen  staff   1.8K Sep 28 20:38 functions.c
-rw-r--r--   1 vallen  staff  234B Sep 28 20:39 functions.h
-rw-r--r--   1 vallen  staff   3.1K Sep 28 20:53 main.c
drwxr-xr-x   2 vallen  staff   64B Sep 28 16:03 testfolder
vallen /Users/vallen/Local Desktop/vscode/6학기 /OS/hw1 20:53:47

```

Figure 4. Example of “ls”, with and without flags, with toggleable features displaying the path and arguments

```

○ (base) vallen@Vallens-MacBook-Pro hw1 % ./app
vallen /Users/vallen/Local Desktop/vscode/6학기 /OS/hw1 20:54:35 pwd

/Users/vallen/Local Desktop/vscode/6학기 /OS/hw1
vallen /Users/vallen/Local Desktop/vscode/6학기 /OS/hw1 20:54:39 ls

Makefile      app      functions.c    functions.h    main.c      testfolder
vallen /Users/vallen/Local Desktop/vscode/6학기 /OS/hw1 20:54:42 cd testfolder

vallen /Users/vallen/Local Desktop/vscode/6학기 /OS/hw1/testfolder 20:54:45 cd ..

vallen /Users/vallen/Local Desktop/vscode/6학기 /OS/hw1 20:54:47 cd ..

vallen /Users/vallen/Local Desktop/vscode/6학기 /OS 20:54:48 pwd

/Users/vallen/Local Desktop/vscode/6학기 /OS
vallen /Users/vallen/Local Desktop/vscode/6학기 /OS 20:54:50

```

Figure 5. Example of directory-related commands (“pwd” and “cd”)

```
vallen /Users/vallen/Local Desktop/vscode/6학기 /OS/hw1 20:57:09 cat functions.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <unistd.h>
#include <time.h>
#include <sys/time.h>
#include <sys/stat.h>
#define MAX_CMD 8
#define MAX_CWD 200

void tokenize_command(char *command_copy, char *command[])
{
    char *checkpoint;
    char *token = strtok_r(command_copy, " ", &checkpoint);
    int loop_counter = 0;
    for(; loop_counter < MAX_CMD && token != NULL; loop_counter++)
    {
        command[loop_counter] = token;
        token = strtok_r(NULL, " ", &checkpoint);
    }
    command[loop_counter] = NULL;
}

void search_command(char *path_copy, char path_buffer[], int limit, char *command[], bool *found) // TODO echo and quit has issues
{
    char *checkpoint;
    char *path_dir = strtok_r(path_copy, ":", &checkpoint);
    struct stat stat_ret;
    while (path_dir != NULL)
    {
        // * Construct the path (add "/" between the directory and the command (executable file))
        snprintf(path_buffer, limit, "%s/%s", path_dir, command[0]);

        // * Check if there is a match, if the match is a regular file, and if the match is an executable file
        if (stat(path_buffer, &stat_ret) == 0 && S_ISREG(stat_ret.st_mode) && (stat_ret.st_mode & S_IXUSR))
        {
            *found = true;
            free(path_copy);
            break;
        }
    }
}
```

Figure 6. Example of “cat” output

```
~
~
~
"example.c" [New] 7L, 73B written
vallen /Users/vallen/Local Desktop/vscode/6학기 /OS/hw1 21:01:58 ls
Makefile      app      example.c      functions.c      functions.h      main.c      testfolder
vallen /Users/vallen/Local Desktop/vscode/6학기 /OS/hw1 21:02:22 gcc example.c -o exampleFile
vallen /Users/vallen/Local Desktop/vscode/6학기 /OS/hw1 21:02:38 ls
Makefile      app      example.c      exampleFile      functions.c      functions.h      main.c      testfolder
vallen /Users/vallen/Local Desktop/vscode/6학기 /OS/hw1 21:02:40 rm example.c exampleFile
vallen /Users/vallen/Local Desktop/vscode/6학기 /OS/hw1 21:03:44 ls
Makefile      app      functions.c      functions.h      main.c      testfolder
vallen /Users/vallen/Local Desktop/vscode/6학기 /OS/hw1 21:03:46
```

Figure 7. Example of “vi”, “gcc”, and “rm” output

```
vallen /Users/vallen/Local Desktop/vscode/6학기 /OS/hw1 21:04:29 echo Hello!
Hello!
vallen /Users/vallen/Local Desktop/vscode/6학기 /OS/hw1 21:05:24 quit

Shell terminated.
○ (base) vallen@Vallens-MacBook-Pro hw1 %
```

Figure 8. Example of “echo” and “quit” output

```
vallen /Users/vallen/Local Desktop/vscode/6학기 /OS/hw1 21:03:44 ls
Makefile      app      functions.c  functions.h  main.c      testfolder
vallen /Users/vallen/Local Desktop/vscode/6학기 /OS/hw1 21:03:46 make clean

rm app
vallen /Users/vallen/Local Desktop/vscode/6학기 /OS/hw1 21:04:28 ls
Makefile      functions.c  functions.h  main.c      testfolder
vallen /Users/vallen/Local Desktop/vscode/6학기 /OS/hw1 21:04:29
```

Figure 9. Example of “make clean” output

```
vallen /Users/vallen/Local Desktop/vscode/6학기 /OS/hw1 21:39:42 ls
Makefile      app      functions.c  functions.h  main.c      testfolder
vallen /Users/vallen/Local Desktop/vscode/6학기 /OS/hw1 21:39:50 cd folder1

Returned an error: No such file or directory
vallen /Users/vallen/Local Desktop/vscode/6학기 /OS/hw1 21:40:08 pwd -x

/bin/pwd: illegal option -- x
usage: pwd [-L | -P]
vallen /Users/vallen/Local Desktop/vscode/6학기 /OS/hw1 21:40:16
```

Figure 10. Example of error messages

Personal Opinion

Overall, I think this project is still relatively simple in terms of implementation, which is great. I did get the proper reminders to get back to coding again after the semester break while also having learned more system-level functions and operations. It also has made me familiar with reading linux man pages for unix-based functions. Apart from that, in regards to the submission, I have also gained experience in simply utilizing git branches.