# Project Report: Unix-Shell

## 1. Introduction

This project aims to develop a background process management tool for Unix-like systems. The purpose of the tool is to provide users with the ability to manage background processes effectively, including listing running and stopped processes, bringing background processes to the foreground, and signalling processes.

## 2. Libraries Included

The project utilizes the following libraries:

* stdio.h: Standard input/output library for basic I/O operations.
* stdlib.h: Standard library for memory allocation and other utility functions.
* string.h: String manipulation library for operations like string copy and comparison.
* unistd.h: Standard symbolic constants and types for system calls.
* sys/types.h: Standard data types for system calls.
* sys/wait.h: Declarations for waiting for process termination.
* signal.h: Defines signal handling functions.
* errno.h: Defines macros for reporting errors.

## 3. Code Explanation

### Jobs.c

#### **init\_list()**

void init\_list()

{

// Initialize background process count and process list

bg\_num\_proc = 0;

proc\_list.num\_proc = 0;

proc\_list.start = NULL;

}

* This function initializes the process list by setting the background process count (bg\_num\_proc) and the number of processes in the list (proc\_list.num\_proc) to zero.
* It also sets the start pointer of the process list (proc\_list.start) to NULL, indicating an empty list

#### **print\_error\_jobs()**

void print\_error\_jobs()

{

// Print usage message with yellow color

print\_YELLOW();

printf("Usage :\n\njobs [options] \n\n");

printf(" -r for running background processes\n\n -s for stopped background process\n\nFlags can be in any order\n\n");

printf(" Multiple flags are handled, in case of illegal flags , apt error is displayed , and jobs is aborted\n\n");

fflush(stdout);

}

This function prints an error message indicating the correct usage of the jobs command.

It provides information about the available options (-r for running background processes and -s for stopped background processes) and their usage.

The message is printed in yellow color using the print\_YELLOW() function.

make\_node()

jobs make\_node(char proc, pid\_t pid, int job\_num)

{

// Allocate memory for a new node

jobs ret\_job = (jobs )malloc(sizeof(jobs));

// Set job number, process name, process ID, and next pointer

ret\_job->job\_num = job\_num;

ret\_job->next = NULL;

strcpy(ret\_job->proc\_name, proc);

ret\_job->proc\_pid = pid;

return ret\_job;

}

* This function creates a new node for the process list.
* It allocates memory for the new node using malloc.
* The function sets the job number, process name, process ID, and next pointer of the new node.
* Finally, it returns a pointer to the newly created node.

#### **add\_node()**

void add\_node(char proc\_name, pid\_t pid)

{

// Increment the number of processes in the list

proc\_list.num\_proc++;

bg\_num\_proc = 0;

// Find the highest job number

for (jobs itr = proc\_list.start; itr != NULL; itr = itr->next)

{

if (bg\_num\_proc < itr->job\_num)

bg\_num\_proc = itr->job\_num;

}

// Increment the job number for the new node

bg\_num\_proc++;

// Create a new node

jobs node = make\_node(proc\_name, pid, bg\_num\_proc);

// Add the new node to the process list

if (proc\_list.start == NULL)

{

proc\_list.start = node;

}

else

{

jobs curr = proc\_list.start, temp = NULL;

// Find the correct position to insert the new node

while (curr != NULL && strcmp(curr->proc\_name, proc\_name) <= 0)

{

temp = curr;

curr = curr->next;

}

if (curr == NULL)

{

temp->next = node;

}

else if (temp == NULL)

{

node->next = proc\_list.start;

proc\_list.start = node;

}

else

{

node->next = temp->next;

temp->next = node;

}

}

}

* This function adds a new node to the process list.
* It first increments the number of processes in the list and finds the highest job number.
* Then, it creates a new node with the given process name, process ID, and job number.
* Next, it finds the correct position to insert the new node based on the lexicographical order of process names.
* Finally, it inserts the new node into the process list.

#### **remove\_node()**

void remove\_node(pid\_t pid)

{

// Check if the process list is empty or if the given PID is invalid

if (proc\_list.num\_proc == 0 || pid <= 0)

return;

// Initialize a pointer to track the previous node

jobs temp = NULL;

// Traverse the process list

for (jobs itr = proc\_list.start; itr != NULL; itr = itr->next)

{

// If the process PID matches the given PID

if (itr->proc\_pid == pid)

{

// If it's the first node in the list

if (temp == NULL)

{

proc\_list.start = itr->next;

free(itr); // Free memory allocated for the node

proc\_list.num\_proc--; // Decrement the number of processes

}

else

{

temp->next = itr->next;

free(itr); // Free memory allocated for the node

proc\_list.num\_proc--; // Decrement the number of processes

}

}

temp = itr; // Update the previous node pointer

}

}

* This function removes a node from the process list based on the given process ID.
* It first checks if the process list is empty or if the given PID is invalid (less than or equal to 0).
* Then, it traverses the process list to find the node with the matching PID.
* If the node is found, it removes the node from the list by updating the pointers and freeing the memory allocated for the node.

**get\_data\_by\_pid()**

jobs get\_data\_by\_pid(pid\_t pid)

{

// Initialize a pointer to store the found node

jobs ret\_data = NULL;

// Traverse the process list

for (jobs itr = proc\_list.start; itr != NULL; itr = itr->next)

{

// If the process PID matches the given PID

if (itr->proc\_pid == pid)

{

ret\_data = itr; // Store the found node

break; // Exit the loop

}

}

return ret\_data; // Return the found node (or NULL if not found)

}

* This function retrieves a node from the process list based on the given process ID.
* It traverses the process list and compares the PID of each node with the given PID.
* If a node with a matching PID is found, it returns a pointer to that node. Otherwise, it returns NULL.

#### **get\_data\_by\_id()**

jobs get\_data\_by\_id(int job\_id)

{

// Initialize a pointer to store the found node

jobs ret\_data = NULL;

// Traverse the process list

for (jobs itr = proc\_list.start; itr != NULL; itr = itr->next)

{

// If the job number matches the given job ID

if (itr->job\_num == job\_id)

{

ret\_data = itr; // Store the found node

break; // Exit the loop

}

}

return ret\_data; // Return the found node (or NULL if not found)

}

* This function retrieves a node from the process list based on the given job number (job ID).
* It traverses the process list and compares the job number of each node with the given job ID.
* If a node with a matching job ID is found, it returns a pointer to that node. Otherwise, it returns NULL.

#### **get\_job\_status()**

char get\_job\_status(pid\_t pid)

{

// Create a string to store the path of the process status file

char proc\_stat\_path[name\_len];

// Construct the path using the process ID

sprintf(proc\_stat\_path, "/proc/%d/status", pid);

// Open the process status file for reading

FILE flag\_stat = fopen(proc\_stat\_path, "r");

// Check if the file opening was successful

if (flag\_stat == NULL)

{

// Print an error message and return NULL if file opening failed

perror(ERROR "Failed to open process status file");

return NULL;

}

// Allocate memory for reading lines from the file

char read\_line = NULL;

size\_t sz\_read = 0;

char ret\_val = NULL; // Initialize a string to store the process status

// Read lines from the file until the end

while (getline(&read\_line, &sz\_read, flag\_stat) != -1)

{

read\_line[strlen(read\_line) - 1] = '\0'; // Remove the newline character

// Tokenize the line based on colon

char temp = strtok(read\_line, ":");

// If the token is "State", retrieve the process status

if (temp != NULL && strcmp(temp, "State") == 0)

{

// Get the next token

temp = strtok(NULL, " \t");

// If the token is not NULL, strdup it (duplicate and return)

if (temp != NULL)

{

ret\_val = strdup(temp); // Duplicate and return the token

break; // Exit the loop

}

}

}

// Free memory allocated for reading lines

* Retrieves a node from the process list based on the given job number (job ID).
* Takes the job number (job ID) of the process to retrieve.
* Returns a pointer to the jobs structure representing the process with the specified job ID or NULL if not found.
* Traverses the process list, comparing job numbers until a match is found or the end of the list is reached.

### Bg.c

#### **execute\_bg()**

void execute\_bg(int arg\_count, char argument[])

{

// Check if the number of arguments is not equal to 2

if (arg\_count != 2)

{

// Print an error message for invalid number of arguments

fprintf(stderr, ERROR "Invalid Number of arguments, correct syntax bg <job\_num>\n" RESET);

return;

}

// Convert the second argument to an integer

int job\_no = atoi(argument[1]);

// Check if the converted job number is less than or equal to 0

if (job\_no <= 0)

{

// Print an error message for invalid job number syntax

fprintf(stderr, ERROR "<job\_number> should be a positive integer, syntax : bg <job\_num>\n" RESET);

return;

}

// Call the bg function with the job number

bg(job\_no);

}

* Checks if the number of arguments is exactly 2.
* Converts the second argument to an integer.
* Verifies if the job number is a positive integer.
* Calls the bg function with the job number.

#### **bg()**

void bg(int job\_num)

{

// Retrieve the job from the process list using the job number

jobs job = get\_data\_by\_id(job\_num);

// Check if the job does not exist

if (job == NULL)

{

// Print an error message for non-existent job number

fprintf(stderr, ERROR "No process exists with given <job\_number>\n" RESET);

return;

}

// Print the job information in yellow and green

print\_YELLOW();

printf("[%d] ", job->job\_num);

print\_GREEN();

printf("%s running in background ", job->proc\_name);

print\_YELLOW();

printf(" [%d]\n", job->proc\_pid);

print\_RESET();

// Send a SIGCONT signal to the process to resume execution

pid\_t proc\_pid = job->proc\_pid;

int flag = kill(proc\_pid, SIGCONT);

// Check if sending the signal failed

if (flag < 0)

{

// Print an error message with perror

perror(ERROR "bg ");

return;

}

}

* Retrieves the job from the process list using the job number.
* Prints the job information in yellow and green.
* Sends a SIGCONT signal to the process to resume execution.
* Checks for errors during signal transmission and prints an error message if needed.

### Cd.c

#### **execute\_cd()**

void execute\_cd(int arg\_count, char argument)

{

if (arg\_count == 1)

cd("~");

else if (arg\_count == 2)

{

cd(argument[1]);

}

else

{

fprintf(stderr, ERROR "TOO MANY ARGUMENTS FOR cd, GIVE ONLY ONE ARGUMENT\n" RESET);

}

}

* This function is responsible for executing the cd command.
* It checks the number of arguments passed to determine the behavior:
* If only one argument is provided, it calls the cd() function with the home directory symbol (~).
* If two arguments are provided, it directly calls the cd() function with the specified path.
* If more than two arguments are provided, it displays an error message indicating too many arguments.

#### **cd()**

void cd(char path)

{

char final\_dir = (char )malloc(name\_len sizeof(char));

if (strcmp(path, "") == 0)

strcpy(final\_dir, cwd\_path);

else if (path[0] == '~')

{

strcpy(final\_dir, get\_home\_dir\_path(path));

}

else if (strcmp(path, "-") == 0)

{

if (strcmp(last\_dir\_visited, "") == 0)

{

fprintf(stderr, ERROR "PWD DIDN'T CHANGED\n" RESET);

free(final\_dir);

return;

}

else

{

printf("%s\n", last\_dir\_visited);

strcpy(final\_dir, last\_dir\_visited);

}

}

else

strcpy(final\_dir, path);

int flag = chdir(final\_dir);

if (flag != 0)

{

perror(ERROR "DIRECTORY CAN'T BE CHANGED ");

print\_RESET();

}

getcwd(last\_dir\_visited, name\_len);

free(final\_dir);

}

* This function changes the current working directory.
* It takes a path as input.
* If the path is an empty string, it copies the current working directory path.
* If the path starts with ~, it retrieves the absolute path of the home directory using the get\_home\_dir\_path() function.
* If the path is -, it checks if the last visited directory is available and changes to that directory.
* Otherwise, it directly copies the provided path.
* It then attempts to change the directory using the chdir() system call.
* If successful, it updates the last visited directory and frees the allocated memory for final\_dir.

### Echo.c

#### **execute\_echo()**

void execute\_echo(int arg\_count, char argument)

{

int i = 1;

while (argument[i] != NULL)

{

printf("%s", argument[i]);

i++;

if (argument[i] != NULL)

printf(" ");

}

printf("\n");

}

* This function executes the echo command.
* It takes the number of arguments and the argument array as input.
* It iterates over the argument array starting from index 1 (skipping the command itself).
* For each argument, it prints the argument followed by a space if there are more arguments to print.
* After printing all arguments, it prints a newline character to terminate the line.

### Execute.c

#### **execute()**

void execute(char command)

{

int len = strlen(command) + 1;

// Check if the command string is null

if (command == NULL)

return;

// Check for pipes in the command and execute piping if found

if (count\_pipes(command) != 0)

{

is\_pipe = true;

execute\_piping(command);

is\_pipe = false;

return;

}

// Check for redirection in the command and execute redirection if found

if (check\_redirection(command) == true)

{

is\_redir = true;

execute\_redirection(command);

is\_redir = false;

return;

}

char str = strtok(command, " ");

//str contains the main command now

// If no main command is found, return

if (str == NULL)

return;

int arg\_count = 0;

char arguments[len];

// Tokenize the command string and store arguments in arguments array

while (str != NULL)

{

arguments[arg\_count] = (char )malloc((strlen(str) + 1) sizeof(char));

strcpy(arguments[arg\_count], str);

arg\_count++;

str = strtok(NULL, " ");

}

arguments[arg\_count] = NULL;

// Execute command based on the main command name

if (strcmp(arguments[0], "ls") == 0)

{

execute\_ls(arg\_count, arguments);

}

else if (strcmp("cd", arguments[0]) == 0)

{

execute\_cd(arg\_count, arguments);

}

else if (strcmp("echo", arguments[0]) == 0)

{

execute\_echo(arg\_count, arguments);

}

else if (strcmp("pwd", arguments[0]) == 0)

{

execute\_pwd();

}

// Check for other built-in commands and execute accordingly

// (history, help, procmon, pinfo, fg, repeat, bg, exit, replay, jobs, sig)

// If not a built-in command, execute as a system command

// (e.g., executing external programs)

// Free dynamically allocated memory for arguments array

for (int i = 0; i < arg\_count; i++)

if (arguments[i] != NULL)

free(arguments[i]);

}

* This function executes the given command.
* It takes the command string as input.
* It first checks for pipes and handles piping if found.
* Then, it checks for redirection and handles redirection if found.
* Next, it tokenizes the command string to extract individual arguments.
* It executes the appropriate built-in command or system command based on the main command name.
* After execution, it frees dynamically allocated memory for the arguments array.

### Fg.c

#### **execute\_fg()**

void execute\_fg(int arg\_count, char argument[])

{

// Check if the number of arguments is correct

if (arg\_count != 2)

{

fprintf(stderr, ERROR "Invalid Number of arguments, correct syntax fg <job\_num>\n" RESET);

return;

}

// Convert the job number argument to integer

int job\_no = atoi(argument[1]);

// Check if the job number is valid

if (job\_no <= 0)

{

fprintf(stderr, ERROR "<job\_number> should be a positive integer, syntax : fg <job\_num>\n" RESET);

return;

}

// Call fg function to bring the specified job into the foreground

fg(job\_no);

}

* This function executes the "fg" command.
* It takes the number of arguments and the array of arguments as input.
* It first checks if the correct number of arguments is provided.
* Then, it converts the job number argument to an integer.
* It checks if the job number is valid.
* Finally, it calls the fg() function to bring the specified job into the foreground.

#### **fg()**

void fg(int job\_num)

{

// Get the process information for the specified job number

jobs proc = get\_data\_by\_id(job\_num);

// Check if process information exists for the given job number

if (proc == NULL)

{

fprintf(stderr, ERROR "No process exists with given <job\_number>\n" RESET);

return;

}

// Retrieve process ID and name from process information

pid\_t proc\_pid = proc->proc\_pid;

char proc\_name[name\_len];

strcpy(proc\_name, proc->proc\_name);

// Bring the process into the foreground

print\_GREEN();

printf("Process %s with pid %d continuing in foreground .... \n", proc\_name, proc\_pid);

print\_RESET();

// Ignore terminal control signals

signal(SIGTTIN, SIG\_IGN);

signal(SIGTTOU, SIG\_IGN);

// Set the process group ID of the terminal to the process group ID of the specified process

tcsetpgrp(0, getpgid(proc\_pid));

// Send SIGCONT signal to the process to continue execution

int x = kill(proc\_pid, SIGCONT);

if (x == -1)

{

perror("");

return;

}

// Wait for the process to finish or be stopped

int w\_st;

waitpid(proc\_pid, &w\_st, WUNTRACED);

fflush(stdout);

// Reset terminal control signals to default behavior

tcsetpgrp(0, getpid());

signal(SIGTTOU, SIG\_DFL);

signal(SIGTTIN, SIG\_DFL);

// If the process is stopped, print a message and remove it from the process list

if (WIFSTOPPED(w\_st))

{

printf("Process %s with pid %d stopped .... \n", proc\_name, proc\_pid);

}

else

remove\_node(proc\_pid);

}

* This function brings the specified job into the foreground.
* It takes the job number as input.
* It retrieves process information for the specified job number.
* It brings the process into the foreground by setting appropriate permissions and sending signals.
* It waits for the process to finish or be stopped.
* If the process is stopped, it prints a message and removes it from the process list.

### Find.c

#### **execute\_find()**

void execute\_find(int arg\_count, char argument[])

{

// Check if the number of arguments is correct

if (arg\_count != 2)

{

fprintf(stderr, "Usage: find <folder-name>\n");

return;

}

// Get the folder name from the arguments

char folder\_name = argument[1];

// Open the root directory

DIR dir = opendir("/");

if (dir == NULL)

{

perror("Error opening root directory");

return;

}

// Read the directory entries

struct dirent entry;

while ((entry = readdir(dir)) != NULL)

{

// Check if the entry is a directory

if (entry->d\_type == DT\_DIR)

{

// Skip "." and ".."

if (strcmp(entry->d\_name, ".") == 0 || strcmp(entry->d\_name, "..") == 0)

{

continue;

}

// Construct the full path of the entry

char path[PATH\_MAX];

snprintf(path, sizeof(path), "/%s", entry->d\_name);

// Get information about the entry

struct stat st;

if (stat(path, &st) == -1)

{

perror("Error stat");

continue;

}

// Check if the entry is a directory

if (S\_ISDIR(st.st\_mode))

{

// Check if the directory name matches the specified folder name

if (strcmp(entry->d\_name, folder\_name) == 0)

{

printf("%s\n", path);

}

// Recursively search in subdirectories

execute\_find\_in\_directory(path, folder\_name);

}

}

}

closedir(dir);

}

* This function searches for a specified folder name in the root directory ("/").
* It takes the number of arguments and an array of arguments as input.
* It checks if the correct number of arguments is provided.
* It opens the root directory ("/") for reading.
* It reads the directory entries and searches for directories.
* If a directory is found, it checks if its name matches the specified folder name.
* If a match is found, it prints the full path of the directory.
* It then recursively searches in subdirectories.

#### **execute\_find\_in\_directory()**

void execute\_find\_in\_directory(const char dir\_path, const char folder\_name)

{

// Open the specified directory

DIR dir = opendir(dir\_path);

if (dir == NULL)

{

perror("Error opening directory");

return;

}

// Read the directory entries

struct dirent entry;

while ((entry = readdir(dir)) != NULL)

{

// Check if the entry is a directory

if (entry->d\_type == DT\_DIR)

{

// Skip "." and ".."

if (strcmp(entry->d\_name, ".") == 0 || strcmp(entry->d\_name, "..") == 0)

{

continue;

}

// Construct the full path of the entry

char path[PATH\_MAX];

snprintf(path, sizeof(path), "%s/%s", dir\_path, entry->d\_name);

// Get information about the entry

struct stat st;

if (stat(path, &st) == -1)

{

perror("Error stat");

continue;

}

// Check if the entry is a directory

if (S\_ISDIR(st.st\_mode))

{

// Check if the directory name matches the specified folder name

if (strcmp(entry->d\_name, folder\_name) == 0)

{

printf("%s\n", path);

}

// Recursively search in subdirectories

execute\_find\_in\_directory(path, folder\_name);

}

}

}

closedir(dir);

}

* This function is a helper function for execute\_find().
* It searches for a specified folder name in a given directory.
* It takes the path of the directory and the folder name to search for as input.
* It opens the specified directory for reading.
* It reads the directory entries and searches for directories.
* If a directory is found, it checks if its name matches the specified folder name.
* If a match is found, it prints the full path of the directory.
* It then recursively searches in subdirectories.

### Help.c

#### **display\_help()**

void display\_help(const char command\_name) {

// Check if a specific command is provided

if (command\_name == NULL) {

// Display list of all available commands

printf("Available commands:\n");

for (size\_t i = 0; i < sizeof(commands) / sizeof(commands[0]); i++) {

printf("%s - %s %s%s\n", PROMPT, commands[i].name, RESET, commands[i].description);

}

} else {

// Find and display help message for the specified command

for (size\_t i = 0; i < sizeof(commands) / sizeof(commands[0]); i++) {

if (strcmp(command\_name, commands[i].name) == 0) {

printf("%sUsage: %s%s\n%s%s\n", PROMPT, commands[i].syntax, RESET, YELLOW, commands[i].description);

return;

}

}

// Command not found

printf("%sCommand '%s' not found. Use 'help' to see available commands.%s\n", ERROR, command\_name, RESET);

}

}

* This function displays help information for a specified command or lists all available commands if no command is provided.
* If no command is provided (command\_name == NULL), it iterates through the commands array and prints the name and description of each command.
* If a command is provided, it searches for the command in the commands array and prints its syntax and description.

#### **help()**

void help(int arg\_count, char argument[]) {

// Check the number of arguments

if (arg\_count == 1) {

// Display general help information

display\_help(NULL);

} else if (arg\_count == 2) {

// Display help for the specified command

display\_help(argument[1]);

} else {

// Invalid number of arguments

printf("%sUsage: help [command]%s\n", ERROR, RESET);

}

}

* This function handles the help command.
* If no arguments are provided, it displays general help information.
* If one argument is provided (the name of a command), it displays help information for that command.
* If more than one argument is provided, it prints an error message indicating the correct usage.

### Helper.c

#### **remove\_char\_front\_and\_back()**

char remove\_char\_front\_and\_back(char input, char ch)

{

// Allocate memory for the result

char ret = (char )malloc(sizeof(input));

int i = 0, len = strlen(input);

// Remove leading characters

while (input[i] == ch)

i++;

// Copy the remaining characters to the result

strcpy(ret, input);

ret[len] = '\0';

for (int itr = strlen(ret) - 1; itr >= 0; itr--)

{

// Remove trailing characters

if (ret[itr] == ch)

ret[itr] = '\0';

}

return ret;

}

* This function removes a specified character from the beginning and end of a string.
* It allocates memory for the result string and iterates through the input string to remove leading and trailing occurrences of the specified character.
* The resulting string is returned.

#### **get\_file\_name\_from\_path()**

char get\_file\_name\_from\_path(char file\_path)

{

if (file\_path == NULL) // Check if the file path is valid

return NULL;

char x = (char )malloc((strlen(file\_path) + 1) sizeof(char));

strcpy(x, file\_path);

char cpy = strstr(x, "/");

if (cpy == NULL) // If the file name itself is provided, it is in the current working directory

return x;

else

{

int lft = 0;

do

{

lft = strlen(cpy) + 1;

x = &x[strlen(x) - lft + 2]; // Rearrange the pointer to the string

cpy = strstr(x, "/"); // Search for the next occurrence of '/'

} while (cpy); // Repeat until cpy becomes null

}

return x;

}

* This function extracts the file name from a given file path.
* It allocates memory for the result string, copies the file path into it, and searches for the last occurrence of '/' to locate the file name.
* The file name is returned.

#### **get\_file\_path()**

char get\_file\_path(char file\_path)

{

if (file\_path == NULL)

return NULL;

char x = (char )malloc((strlen(file\_path) + 1) sizeof(char));

strcpy(x, file\_path);

char fileName = get\_file\_name\_from\_path(x);

char chk = strstr(x, fileName);

if (chk)

{

strcpy(chk, "");

return x;

}

else

return NULL;

}

* This function retrieves the directory path from a given file path.
* It allocates memory for the result string, copies the file path into it, and uses get\_file\_name\_from\_path() to obtain the file name.
* It then removes the file name portion from the file path to extract the directory path.

#### **get\_home\_dir\_path()**

char get\_home\_dir\_path(char path)

{

// Allocate memory for the result string

char ret\_val = (char )malloc(strlen(cwd\_path) + strlen(path) + 7);

// Construct the home directory path

sprintf(ret\_val, "%s/%s", cwd\_path, &path[1]);

return ret\_val;

}

* This function constructs the absolute path of a file located in the home directory.
* It allocates memory for the result string and uses sprintf to concatenate the current working directory path (cwd\_path) with the provided file path.
* The constructed path is returned.
* print\_RED(), print\_GREEN(), print\_YELLOW(), print\_WHITE(), print\_MAGENTA(), print\_RESET(), print\_BLUE()
* These functions are used to print colored text to the terminal, depending on whether the shell is currently processing redirection or piping. If neither redirection nor piping is active, these functions print colored text as specified in the shell's header file.

### Pinfo.c

#### **execute\_pinfo()**

void execute\_pinfo(int arg\_count, char argument[])

{

pid\_t pid\_process;

// Determine the process ID to get information about

if (arg\_count > 2)

{

// Print error message for invalid number of arguments

fprintf(stderr, ERROR "INVALID NUMBER OF ARGUMENTS , GIVE SYNTAX OF THE TYPE PINFO <job\_number>\n" RESET);

return;

}

else if (arg\_count == 1)

{

pid\_process = SHELL\_PID; // Set process ID to shell's process ID if no argument provided

}

else

{

// Convert argument to process ID

pid\_process = strtol(argument[1], NULL, 10);

// Print error message for negative process ID

if (pid\_process < 0)

{

fprintf(stderr, ERROR "JOB NUMBER CAN'T BE A NEGATIVE NUMBER\n" RESET);

return;

}

}

char proc\_stat\_path[name\_len], proc\_exe\_path[name\_len];

// Construct paths for process status and executable

sprintf(proc\_stat\_path, "/proc/%d/status", pid\_process);

sprintf(proc\_exe\_path, "/proc/%d/exe", pid\_process);

FILE flag\_stat = fopen(proc\_stat\_path, "r");

// Open process status file

if (flag\_stat == NULL)

{

// Print error message if file opening fails

perror(ERROR "pinfo ");

fprintf(stderr, RESET);

return;

}

int itr = 0;

char read\_line = NULL;

size\_t sz\_read = 0;

proc\_info st\_proc;

st\_proc.proc\_pid = pid\_process;

// Read lines from process status file

while (getline(&read\_line, &sz\_read, flag\_stat) != -1)

{

itr++;

read\_line[strlen(read\_line) - 1] = '\0';

// Parse process status and virtual memory size

if (itr == 3)

{

char temp = strtok(read\_line, ":");

temp = strtok(NULL, " \t");

if (temp != NULL)

strcpy(st\_proc.proc\_status, temp);

else

st\_proc.proc\_status[0] = '\0';

}

else if (itr == 18)

{

char temp = strtok(read\_line, ":");

temp = strtok(NULL, " \t");

if (temp != NULL)

{

strcpy(st\_proc.virtual\_mem\_size, temp); // Store virtual memory size

temp = strtok(NULL, " \t");

if (temp != NULL)

{

char tmp[name\_len];

sprintf(tmp, " %s { Virtual Memory }", temp);

strcat(st\_proc.virtual\_mem\_size, tmp);

}

}

break;

}

free(read\_line);

read\_line = NULL;

}

fclose(flag\_stat);

size\_t flag\_exe = readlink(proc\_exe\_path, read\_line, name\_len);

// Read symbolic link to get executable path

if (flag\_exe < 0)

{

// Print error message if reading symbolic link fails

perror(ERROR "COULD NOT FETCH THE EXECUTABLE ");

fprintf(stderr, RESET);

return;

}

read\_line[flag\_exe] = '\0';

// Determine the executable path relative to the current working directory

if (strcmp(read\_line, cwd\_path) == 0)

{

strcpy(st\_proc.exec\_path, "~");

}

else if (strlen(read\_line) < strlen(cwd\_path))

{

strcpy(st\_proc.exec\_path, read\_line);

}

else

{

if (strstr(read\_line, cwd\_path) != NULL)

{

int len = strlen(cwd\_path);

char rel\_dir = (char )malloc((strlen(read\_line) - len + 1) sizeof(char));

strcpy(rel\_dir, &read\_line[len]);

char str[name\_len];

sprintf(str, "~%s", rel\_dir);

strcpy(st\_proc.exec\_path, str);

free(rel\_dir);

}

else

{

strcpy(st\_proc.exec\_path, read\_line);

}

}

if (pid\_process == SHELL\_PID)

strcat(st\_proc.proc\_status, "+");

// Print process information

print\_WHITE();

printf("PID-- %d\nPROCESS STATUS -- %s\nMEMORY -- %s\nEXECUTABLE PATH -- %s\n", st\_proc.proc\_pid, st\_proc.proc\_status, st\_proc.virtual\_mem\_size, st\_proc.exec\_path);

print\_RESET();

}

* Determine the process ID to retrieve information about based on the provided arguments.
* Open the process status file and parse its contents to obtain process status and virtual memory size.
* Read the symbolic link of the process executable to determine its path.
* Determine the relative path of the executable with respect to the current working directory.
* Print the retrieved process information, including PID, process status, memory usage, and executable path.

### Pipe.c

#### **execute\_piping()**

void execute\_piping(char command)

{

// Count the number of pipes in the command

int num\_pipes = count\_pipes(command);

int num\_pipe\_commands = num\_pipes + 1;

// Duplicate standard input and output file descriptors

int stdin\_copy = dup(STDIN\_FILENO);

int stdout\_copy = dup(STDOUT\_FILENO);

// Allocate memory for pipe arguments and parse the command

char pipe\_arguments[num\_pipe\_commands + 1];

int arg\_count = 0;

char str = strtok(command, "|");

// Execute commands with piping

for (int itr = 0; itr < arg\_count - 1 && pipe\_arguments[itr] != NULL; itr++)

{

// Establish pipes between processes

int flag\_pipe = pipe(proc\_pipe[itr]);

// Redirect input and output for piped commands

int flag\_dup = dup2(proc\_pipe[itr - 1][0], STDIN\_FILENO);

int flag\_dup2 = dup2(proc\_pipe[itr][1], STDOUT\_FILENO);

// Execute the piped command

execute(pipe\_arguments[itr]);

// Close pipe file descriptors

close(proc\_pipe[itr - 1][0]);

close(proc\_pipe[itr][1]);

}

// Reset input and output file descriptors

int flag\_dup = dup2(proc\_pipe[itr - 1][0], STDIN\_FILENO);

int flag\_dup2 = dup2(stdout\_copy, STDOUT\_FILENO);

// Reset the pipe flag and execute the final command

is\_pipe = false;

execute(pipe\_arguments[itr]);

// Restore standard input

int fg = dup2(stdin\_copy, STDIN\_FILENO);

// Close file descriptors and free memory

close(proc\_pipe[itr - 1][0]);

close(stdin\_copy);

close(stdout\_copy);

free\_pointer(arg\_count, pipe\_arguments);

}

* Count the number of pipes in the command string.
* Duplicate standard input and output file descriptors to preserve them.
* Parse the command string and allocate memory for pipe arguments.
* Execute commands with piping by establishing pipes between processes and redirecting input and output.
* Reset file descriptors after executing piped commands.
* Restore standard input and close file descriptors and free memory.

### Procmon.c

#### **execute\_procmon()**

void execute\_procmon() {

// Open the /proc directory

DIR proc\_dir = opendir("/proc");

// Traverse the /proc directory and print process IDs and names

while ((entry = readdir(proc\_dir)) != NULL) {

if (entry->d\_type == DT\_DIR && atoi(entry->d\_name) != 0) {

// Construct the path to the process status file

snprintf(status\_path, sizeof(status\_path), "/proc/%s/status", entry->d\_name);

// Open the process status file

status\_file = fopen(status\_path, "r");

// Read the process name from the status file

while (fgets(line, sizeof(line), status\_file) != NULL) {

if (strncmp(line, "Name:", 5) == 0) {

// Print the process ID and process name

printf("%s\t%s", entry->d\_name, line + 6);

break;

}

}

fclose(status\_file);

printf("\n");

}

}

// Close the /proc directory

closedir(proc\_dir);

}

* Open the /proc directory.
* Traverse the /proc directory and print process IDs and names.
* Construct the path to the process status file and open it.
* Read the process name from the status file and print it.
* Close the process status file and the /proc directory.

### History.c

#### **init\_history()**

void init\_history()

{

// Open history file for reading

FILE hist\_file = fopen(HISTORY\_FILE\_NAME, "r");

hist\_count = 0;

last\_hist\_entry\_added = 0;

if (hist\_file == NULL)

{

hist\_file = fopen(HISTORY\_FILE\_NAME, "w"); // Create the file if it doesn't exist

fclose(hist\_file);

return;

}

char input = NULL;

size\_t sz\_line = 0;

// Read history entries from file

while (getline(&input, &sz\_line, hist\_file) != -1)

{

if (input[strlen(input) - 1] == '\n')

input[strlen(input) - 1] = '\0';

// Copy command to history list

strcpy(history\_list[hist\_count].command, input);

hist\_count++;

}

last\_hist\_entry\_added = hist\_count - 1;

int x = fclose(hist\_file);

if (x < 0)

{

fprintf(stderr, ERROR "UNABLE TO LOAD HISTORY\n" RESET);

}

}

* This function initializes the command history by reading entries from a history file.
* It opens the history file for reading and reads each line (command) from the file.
* Each command is copied to the history\_list array, and the total history count is updated.
* If the history file doesn't exist, it creates an empty file.

#### **add\_to\_history()**

void add\_to\_history(char command)

{

if (command == NULL)

return;

if (hist\_count == 0)

{

strcpy(history\_list[hist\_count].command, command);

last\_hist\_entry\_added++;

hist\_count++;

return;

}

int new\_idx = hist\_count % 20;

if (strcmp(history\_list[last\_hist\_entry\_added].command, command) != 0)

{

strcpy(history\_list[new\_idx].command, command);

last\_hist\_entry\_added = new\_idx;

hist\_count++;

return;

}

}

* This function adds a command to the command history.
* It checks if the history is empty. If so, it adds the command directly to the history list.
* If not, it calculates the index for the new command based on the total history count and adds the command to the list if it's different from the last command.

#### **write\_back\_history()**

void write\_back\_history()

{

FILE hist\_file = fopen(HISTORY\_FILE\_NAME, "w");

if (hist\_file == NULL)

{

fprintf(stderr, ERROR);

fprintf(stderr, RESET);

return;

}

int sz\_history = 20;

if (hist\_count < 20)

sz\_history = hist\_count;

if (hist\_count >= 20)

{

int start\_idx = hist\_count % 20;

for (int i = start\_idx; i < sz\_history; i++)

fprintf(hist\_file, "%s\n", history\_list[i].command);

for (int i = 0; i < start\_idx; i++)

fprintf(hist\_file, "%s\n", history\_list[i].command);

}

else

{

for (int i = 0; i < sz\_history; i++)

fprintf(hist\_file, "%s\n", history\_list[i].command);

}

int x = fclose(hist\_file);

if (x < 0)

{

fprintf(stderr, ERROR "UNABLE TO WRITE\_BACK HISTORY\n" RESET);

}

}

* This function writes the command history back to the history file.
* It opens the history file for writing and writes each command from the history list to the file.
* If the history count is less than 20, it writes all history entries. Otherwise, it writes the most recent 20 entries.
* The function then closes the history file.

#### **history()**

void history(int arg\_count, char argument[])

{

if (arg\_count > 2)

{

fprintf(stderr, ERROR "history : TOO MANY ARGUMENTS\n" RESET);

return;

}

if (arg\_count == 1)

{

int sz\_history = 10, cnt = 0;

if (hist\_count < 10)

{

sz\_history = hist\_count;

if (hist\_count == 0)

{

fprintf(stderr, ERROR "EMPTY HISTORY NOTHING TO SHOW.\n" RESET);

return;

}

}

for (int i = last\_hist\_entry\_added;; i--)

{

if (i < 0)

i = sz\_history - 1;

print\_WHITE();

printf("[%d] ", cnt + 1);

print\_YELLOW();

printf(": %s\n", history\_list[i].command);

cnt++;

if (cnt == sz\_history)

break;

}

}

else

{

int idx = atoi(argument[1]);

if (idx <= 0)

{

fprintf(stderr, ERROR "history : PROVIDE A POSITIVE INTEGER AS ARGUMENT\n" RESET);

return;

}

int cnt = 0;

int sz\_history = 20;

if (hist\_count < 20)

{

sz\_history = hist\_count;

if (hist\_count == 0)

{

fprintf(stderr, ERROR "EMPTY HISTORY NOTHING TO SHOW\n" RESET);

return;

}

if (idx > hist\_count)

{

fprintf(stderr, ERROR "HISTORY ONLY HAS %d ENTRIES TO SHOW, DISPLAYING THEM\n" RESET, hist\_count);

idx = hist\_count;

}

}

if (idx > 20)

{

fprintf(stderr, ERROR "HISTORY ONLY HAS 20 ENTRIES TO SHOW, DISPLAYING THEM\n" RESET);

idx = 20;

}

int start\_idx = hist\_count % 20;

for (int i = last\_hist\_entry\_added;; i--)

{

if (i < 0)

i = sz\_history - 1;

print\_WHITE();

printf("[%d] ", cnt + 1);

print\_YELLOW();

printf(": %s\n", history\_list[i].command);

cnt++;

if (cnt == idx)

break;

}

}

}

* This function displays the command history.
* If no argument is provided, it displays the last 10 history entries.
* If an argument (a positive integer) is provided, it displays that number of history entries.
* The function handles cases where the number of history entries to display exceeds the total history count.

### Input.c

#### **getInput()**

char getInput()

{

char input = NULL;

unsigned long int sz\_input = 0;

ssize\_t r = getline(&input, &sz\_input, stdin);

if (r > 0)

{

// Remove trailing newline character if present

if (input[strlen(input) - 1] == '\n')

input[strlen(input) - 1] = '\0';

}

else

execute("exit"); // If no input is received, exit the shell

return input;

}

* This function reads input from the standard input (stdin) using the getline() function.
* It dynamically allocates memory for the input buffer.
* If input is received successfully, it removes the trailing newline character (if present) from the input string.
* If no input is received (e.g., end-of-file is reached), it executes the "exit" command to terminate the shell.
* Finally, it returns the input string to the caller.

### Ls.c

#### **not\_hidden()**

bool not\_hidden(const struct dirent s)

{

if (strlen(s->d\_name) > 0 && s->d\_name[0] == '.')

{

return false;

}

return true;

}

#### **get\_file\_permissions()**

char get\_file\_permissions(char file\_path)

{

struct stat st;

int flag = stat(file\_path, &st);

if (flag < 0)

{

perror(ERROR "FAILED TO FETCH FILE PERMISSIONS ");

fprintf(stderr, RESET);

return NULL;

}

char permissions = (char )malloc(sizeof(char) (10 + 1));

int i = 0;

if (flag == 0 && S\_ISDIR(st.st\_mode))

permissions[i++] = 'd';

else

permissions[i++] = '-';

if (st.st\_mode & S\_IRUSR)

permissions[i++] = 'r';

else

permissions[i++] = '-';

if (st.st\_mode & S\_IWUSR)

permissions[i++] = 'w';

else

permissions[i++] = '-';

if (st.st\_mode & S\_IXUSR)

permissions[i++] = 'x';

else

permissions[i++] = '-';

if (st.st\_mode & S\_IRGRP)

permissions[i++] = 'r';

else

permissions[i++] = '-';

if (st.st\_mode & S\_IWGRP)

permissions[i++] = 'w';

else

permissions[i++] = '-';

if (st.st\_mode & S\_IXGRP)

permissions[i++] = 'x';

else

permissions[i++] = '-';

if (st.st\_mode & S\_IROTH)

permissions[i++] = 'r';

else

permissions[i++] = '-';

if (st.st\_mode & S\_IWOTH)

permissions[i++] = 'w';

else

permissions[i++] = '-';

if (st.st\_mode & S\_IXOTH)

permissions[i++] = 'x';

else

permissions[i++] = '-';

permissions[10] = '\0';

return permissions;

}

#### **execute\_ls()**

void execute\_ls(int arg\_count, char argument[])

{

a\_flag = false;

l\_flag = false;

num\_dir\_ls = 0;

for (int i = 1; i < arg\_count; i++)

{

if (argument[i] != NULL && argument[i][0] == '-')

{

if (argument[i][1] != '\0')

{

int itr = 1;

while (argument[i][itr] != '\0')

{

if (argument[i][itr] == 'a')

a\_flag = true;

if (argument[i][itr] == 'l')

l\_flag = true;

itr++;

}

}

}

}

char list\_dir[arg\_count][name\_len];

for (int i = 1; i < arg\_count; i++)

{

if (argument[i] != NULL && argument[i][0] != '-')

{

strcpy(list\_dir[num\_dir\_ls], argument[i]);

num\_dir\_ls++;

}

else if (argument[i] != NULL && strcmp(argument[i], "-") == 0)

{

strcpy(list\_dir[num\_dir\_ls], argument[i]);

num\_dir\_ls++;

}

}

if (num\_dir\_ls == 0)

{

getcwd(list\_dir[0], name\_len);

if (list\_dir[0] == NULL)

{

perror(ERROR "");

fprintf(stderr, RESET);

return;

}

num\_dir\_ls++;

}

for (int i = 0; i < num\_dir\_ls; i++)

{

ls(list\_dir[i]);

}

}

#### **ls()**

void ls(char file\_path)

{

if (l\_flag == false)

{

char path[name\_len + 10];

if (file\_path[0] != '\0' && file\_path[0] == '~')

{

sprintf(path, "%s/%s", cwd\_path, &file\_path[1]);

}

else if (file\_path[0] != '\0')

{

strcpy(path, file\_path);

}

struct stat st;

int flag = stat(path, &st);

if (flag < 0)

{

perror(ERROR "");

fprintf(stderr, RESET);

return;

}

struct dirent fs;

int num\_file;

if (S\_ISDIR(st.st\_mode))

{

if (a\_flag == true)

num\_file = scandir(path, &fs, NULL, alphasort);

else

num\_file = scandir(path, &fs, not\_hidden, alphasort);

if (num\_file < 0)

{

perror(ERROR "");

fprintf(stderr, RESET);

if (fs != NULL)

free(fs);

}

if (num\_dir\_ls > 1)

{

print\_WHITE();

printf("\n%s :\n", file\_path);

print\_RESET();

}

for (int i = 0; i < num\_file; i++)

{

if (fs[i]->d\_type == 4)

print\_BLUE();

else if (fs[i]->d\_type == 8)

print\_YELLOW();

printf("%s\n", fs[i]->d\_name);

free(fs[i]);

}

free(fs);

}

else

{

print\_YELLOW();

printf("%s\n", path);

print\_RESET();

}

}

else

{

char path[name\_len + 10];

if (file\_path[0] != '\0' && file\_path[0] == '~')

{

sprintf(path, "%s/%s", cwd\_path, &file\_path[1]);

}

else if (file\_path[0] != '\0')

{

strcpy(path, file\_path);

}

struct stat st;

int flag = stat(path, &st);

if (flag < 0)

{

perror(ERROR "ls");

fprintf(stderr, RESET);

return;

}

struct dirent fs;

int num\_file;

if (S\_ISDIR(st.st\_mode))

{

if (a\_flag == true)

num\_file = scandir(path, &fs, NULL, alphasort);

else

num\_file = scandir(path, &fs, not\_hidden, alphasort);

if (num\_file < 0)

{

perror(ERROR "ls");

fprintf(stderr, RESET);

if (fs != NULL)

free(fs);

}

if (num\_dir\_ls > 1)

{

print\_WHITE();

printf("\n%s:\n", file\_path);

print\_RESET();

}

blkcnt\_t total\_blocks = 0;

for (int j = 0; j < num\_file; j++)

{

char dir[2 name\_len];

sprintf(dir, "%s/%s", path, fs[j]->d\_name);

struct stat st\_tmp;

int x = stat(dir, &st\_tmp);

if (x == 0)

{

total\_blocks += st\_tmp.st\_blocks;

}

}

print\_WHITE();

printf("\ntotal : %ld\n", total\_blocks / 2);

print\_RESET();

for (int j = 0; j < num\_file; j++)

{

char dir[2 name\_len];

sprintf(dir, "%s/%s", path, fs[j]->d\_name);

struct stat st\_tmp;

int x = stat(dir, &st\_tmp);

if (x < 0)

{

perror(ERROR "ls");

fprintf(stderr, RESET);

continue;

}

else

{

char file\_permissions = get\_file\_permissions(dir);

struct group grp = getgrgid(st\_tmp.st\_gid);

struct passwd pw = getpwuid(st\_tmp.st\_uid);

struct tm file\_time = localtime(&st\_tmp.st\_mtime);

nlink\_t file\_hlink = st\_tmp.st\_nlink;

char time\_lst\_modified = (char )malloc(sizeof(char) 100);

bool flag\_year = false;

if (file\_time->tm\_year == SHELL\_YEAR && SHELL\_MONTH - file\_time->tm\_mon < 6)

flag\_year = false;

else if (file\_time->tm\_year == SHELL\_YEAR && SHELL\_MONTH - file\_time->tm\_mon >= 6)

flag\_year = true;

else

{

int num\_year = SHELL\_YEAR - file\_time->tm\_year;

if (num\_year > 1)

flag\_year = true;

else

{

int mon = 11 - file\_time->tm\_mon + SHELL\_MONTH;

if (mon > 4)

flag\_year = true;

}

}

if (flag\_year == false)

strftime(time\_lst\_modified, 100, "%b %d %H:%M", file\_time);

else

strftime(time\_lst\_modified, 100, "%b %d %Y", file\_time);

printf("%s\t%4ld\t%s\t%s\t%9ld\t%s\t", file\_permissions, st\_tmp.st\_nlink, grp->gr\_name, pw->pw\_name, st\_tmp.st\_size, time\_lst\_modified);

if (file\_permissions[0] == 'd')

{

print\_BLUE();

printf("%s\n", fs[j]->d\_name);

print\_RESET();

}

else if (file\_permissions[3] == 'x')

{

print\_MAGENTA();

printf("%s\n", fs[j]->d\_name);

print\_RESET();

}

else

{

print\_YELLOW();

printf("%s\n", fs[j]->d\_name);

print\_RESET();

}

free(fs[j]);

free(file\_permissions);

free(time\_lst\_modified);

}

}

free(fs);

}

else

{

char file\_permissions = get\_file\_permissions(path);

struct group grp = getgrgid(st.st\_gid);

struct tm file\_time = localtime(&st.st\_mtime);

struct passwd pw = getpwuid(st.st\_uid);

nlink\_t file\_hlink = st.st\_nlink;

char time\_lst\_modified = (char )malloc(sizeof(char) 100);

bool flag\_year = false;

if (file\_time->tm\_year == SHELL\_YEAR && SHELL\_MONTH - file\_time->tm\_mon < 6)

flag\_year = false;

else if (file\_time->tm\_year == SHELL\_YEAR && SHELL\_MONTH - file\_time->tm\_mon >= 6)

flag\_year = true;

else

{

int num\_year = SHELL\_YEAR - file\_time->tm\_year;

if (num\_year > 1)

flag\_year = true;

else

{

int mon = 11 - file\_time->tm\_mon + SHELL\_MONTH;

if (mon > 4)

flag\_year = true;

}

}

if (flag\_year == false)

strftime(time\_lst\_modified, 100, "%b %d %H:%M", file\_time);

else

strftime(time\_lst\_modified, 100, "%b %d %Y", file\_time);

printf("%s\t%4ld\t%s\t%s\t%9ld\t%s\t", file\_permissions, st.st\_nlink, grp->gr\_name, pw->pw\_name, st.st\_size, time\_lst\_modified);

if (file\_permissions[3] == 'x')

{

print\_MAGENTA();

printf("%s\n", file\_path);

print\_RESET();

}

else

{

print\_YELLOW();

printf("%s\n", file\_path);

print\_RESET();

}

free(file\_permissions);

free(time\_lst\_modified);

}

}

* This file implements the ls command functionality.
* The execute\_ls function parses command-line arguments to determine the flags (a for all and l for long format) and the directories/files to list.
* The ls function takes a file path as input and lists the contents of the directory specified by the path or details about the file if it's a regular file.
* It retrieves file permissions using the get\_file\_permissions function, which dynamically allocates memory for the permissions string.
* The function also retrieves other information like file size, owner, group, and last modified time to display in long format.
* Colors are used to highlight different types of files and directories for better visualization.
* The not\_hidden function filters out hidden files and directories when the -a flag is not provided.
* Error handling is done for file and directory access.

### Main.c

#### **main()**

int main()

{

is\_fg = -1; // Flag to track foreground/background execution

init\_shell(); // Initialize the shell environment

init\_history(); // Initialize command history

// Register signal handlers for child processes, interrupt, and stop signals

signal(SIGCHLD, SIGCHILD\_HANDLER);

signal(SIGINT, SIGINT\_HANDLER);

signal(SIGTSTP, SIGTSTP\_HANDLER);

char in; // Input string

char list; // Tokenized command list

prompt(); // Display initial shell prompt

// Duplicate standard input and output file descriptors

copy\_stdin\_fileno = dup(STDIN\_FILENO);

copy\_stdout\_fileno = dup(STDOUT\_FILENO);

// Main shell loop

while (1)

{

// Get input from user

in = getInput();

char command[name\_len];

strcpy(command, in);

// Add input command to history

add\_to\_history(command);

// Tokenize input command

list = getList(command);

list[num\_command] = NULL;

// Execute each tokenized command

for (int i = 0; list != NULL && list[i] != NULL && i < num\_command; i++)

{

execute(list[i]);

}

// Free allocated memory

free(in);

free(list);

// Display shell prompt

prompt();

}

// Execute exit command (should never reach here)

execute("exit");

}

* Header Includes: It includes various header files required for the shell's functionality.
* Main Function: The entry point of the program. It initializes the shell, sets up signal handlers, initializes command history, and enters an infinite loop to continuously prompt for and execute commands.
* Shell Initialization: Calls init\_shell() to initialize the shell environment.
* Signal Handling: Registers signal handlers for SIGCHLD, SIGINT, and SIGTSTP signals.
* Input Processing: Gets input from the user using getInput() function and adds the input to the command history using add\_to\_history().
* Tokenization: Tokenizes the input command into individual tokens using getList() function.
* Command Execution: Executes each tokenized command using execute() function.
* Prompting: Displays the shell prompt using prompt() function.
* Cleanup: Frees allocated memory and repeats the loop until the exit command is executed.

### Prompt.c

#### **init\_shell()**

void init\_shell()

{

// Clear the terminal and print welcome message

system("clear");

printf("\e[32m\t\tWELCOME TO UNIX-SHELL\e[0m\n");

printf("\e[35m\tA Project by Mustafa, Laiba & Umair\e[0m\n");

// Get host name, user name, current working directory, and shell process ID

gethostname(HOSTNAME, name\_len);

strcpy(USERNAME, getenv("USERNAME"));

getcwd(cwd\_path, name\_len);

SHELL\_PID = getpid();

strcpy(last\_dir\_visited, "");

// Get current date and time

time\_t temp;

time(&temp);

struct tm login\_time = localtime(&temp);

// Set shell month and year

SHELL\_MONTH = login\_time->tm\_mon;

sprintf(HISTORY\_FILE\_NAME, "%s/history.txt", cwd\_path);

SHELL\_YEAR = login\_time->tm\_year;

// Initialize job list

init\_list();

is\_cwd = 1;

}

#### **prompt()**

void prompt()

{

char dir[name\_len];

getcwd(dir, name\_len);

// Check if current directory is the shell directory or not

int path = strcmp(dir, cwd\_path);

if (path == 0)

is\_cwd = 1;

else

is\_cwd = 0;

// Print username, hostname, and current directory

print\_GREEN();

printf("<%s@%s:", USERNAME, HOSTNAME);

print\_RESET();

// Print directory based on various conditions

if (is\_cwd == 1)

{

print\_BLUE();

printf("~");

print\_RESET();

print\_GREEN();

printf("> ");

print\_RESET();

}

else if (strlen(dir) < strlen(cwd\_path))

{

print\_BLUE();

printf("%s", dir);

print\_RESET();

print\_GREEN();

printf("> ");

print\_RESET();

}

else

{

if (strstr(dir, cwd\_path) != NULL)

{

// Print abbreviated directory path

int len = strlen(cwd\_path);

char rel\_dir = (char )malloc((strlen(dir) - len + 1) sizeof(char));

strcpy(rel\_dir, &dir[len]);

print\_BLUE();

printf("~%s", rel\_dir);

print\_RESET();

print\_GREEN();

printf("> ");

print\_RESET();

free(rel\_dir);

}

else

{

// Print full directory path

print\_BLUE();

printf("%s", dir);

print\_RESET();

print\_GREEN();

printf("> ");

print\_RESET();

}

}

}

* init\_shell(): Initializes the shell environment, clears the terminal, prints welcome message, sets up host name, user name, current working directory, shell process ID, and initializes the job list.
* prompt(): Displays the shell prompt with username, hostname, and current directory. Handles different cases for displaying the current directory, including the shell directory and abbreviated paths.

### Pwd.c

#### **execute\_pwd()**

void execute\_pwd()

{

// Allocate memory for current working directory

char cwd = (char )malloc(sizeof(char) name\_len);

// Get current working directory

getcwd(cwd, name\_len);

// Print current working directory

if (cwd != NULL)

{

printf("%s\n", cwd);

free(cwd);

}

else

{

// Print error message if getting current working directory fails

perror(ERROR " ERROR GETTING CWD ");

fprintf(stderr, RESET);

}

}

* execute\_pwd(): Prints the current working directory. It allocates memory to store the path, retrieves the current working directory, and prints it. If getting the current working directory fails, it prints an error message.

### Redirection.c

#### **check\_redirection()**

int check\_redirection(char comm)

{

// Check if input command contains redirection symbols

char input\_redirection = strstr(comm, "<");

char output\_redirection = strstr(comm, ">");

return (input\_redirection != NULL || output\_redirection != NULL);

}

#### **free\_ptrs()**

void free\_ptrs(int arg\_count, char arguments[])

{

// Free memory allocated for command arguments

for (int i = 0; i < arg\_count; i++)

if (arguments[i] != NULL)

free(arguments[i]);

}

#### **execute\_redirection()**

void execute\_redirection(char comm)

{

// Initialize redirection flags and variables

bool in = false, in\_file = false;

bool out = false, out\_file = false;

bool append = false;

char input\_file[name\_len];

char output\_file[name\_len];

int fd\_in, fd\_out;

input\_file[0] = '\0', output\_file[0] = '\0';

// Split command into arguments

size\_t arg\_len = strlen(comm) + 1;

char str = strtok(comm, " ");

if (str == NULL)

return;

char arguments[arg\_len];

int arg\_count = 0;

// Copy arguments into array

while (str != NULL)

{

// Check for input redirection symbol

if (strcmp(str, "<") == 0)

{

// Handle input redirection

if (in || out)

{

fprintf(stderr, ERROR "shell : invalid redirection syntax unexpected token near <\n" RESET);

free\_ptrs(arg\_count, arguments);

return;

}

in = true;

in\_file = true;

str = strtok(NULL, " ");

continue;

}

// Check for output redirection symbol

else if (strcmp(">", str) == 0)

{

// Handle output redirection

if (in || out)

{

fprintf(stderr, ERROR "shell : invalid redirection syntax unexpected token near >\n" RESET);

free\_ptrs(arg\_count, arguments);

return;

}

if (append)

append = false;

out = true;

out\_file = true;

str = strtok(NULL, " ");

continue;

}

// Check for append redirection symbol

else if (strcmp(">>", str) == 0)

{

// Handle append redirection

if (in || out)

{

fprintf(stderr, ERROR "shell : invalid redirection syntax unexpected token near >>\n" RESET);

free\_ptrs(arg\_count, arguments);

return;

}

out\_file = true;

out = true;

append = true;

str = strtok(NULL, " ");

continue;

}

// Handle arguments

if (in == true)

{

strcpy(input\_file, str);

in = false;

str = strtok(NULL, " ");

continue;

}

else if (out == true)

{

strcpy(output\_file, str);

out = false;

str = strtok(NULL, " ");

continue;

}

arguments[arg\_count] = (char )malloc((strlen(str) + 1) sizeof(char));

strcpy(arguments[arg\_count], str);

arg\_count++;

str = strtok(NULL, " ");

}

arguments[arg\_count] = NULL;

// Check for invalid redirection syntax

if (in || out)

{

fprintf(stderr, ERROR "shell : invalid redirection syntax\n" RESET);

free\_ptrs(arg\_count, arguments);

return;

}

// Perform input redirection if specified

if (in\_file == true)

{

if (input\_file[0] == '\0')

{

fprintf(stderr, ERROR "shell : no file given for input redirection\n" RESET);

free\_ptrs(arg\_count, arguments);

return;

}

fd\_in = open(input\_file, O\_RDONLY);

if (fd\_in <= 0)

{

perror(ERROR "shell : couldn't redirect, invalid input file\n");

free\_ptrs(arg\_count, arguments);

return;

}

int flag = dup2(fd\_in, STDIN\_FILENO);

if (flag < 0)

{

perror(ERROR "shell : couldn't redirect, invalid input file\n");

free\_ptrs(arg\_count, arguments);

close(fd\_in);

return;

}

}

// Perform output redirection if specified

if (out\_file == true)

{

if (output\_file[0] == '\0')

{

fprintf(stderr, ERROR "shell : no file given for output redirection\n" RESET);

free\_ptrs(arg\_count, arguments);

if (input\_file)

close(fd\_in);

return;

}

if (append == true)

fd\_out = open(output\_file, O\_WRONLY | O\_CREAT | O\_APPEND, 0644);

else

fd\_out = open(output\_file, O\_WRONLY | O\_CREAT | O\_TRUNC, 0644);

if (fd\_out <= 0)

{

perror(ERROR "shell : couldn't redirect, invalid output file ");

if (input\_file)

close(fd\_in);

free\_ptrs(arg\_count, arguments);

return;

}

int flag = dup2(fd\_out, STDOUT\_FILENO);

if (flag < 0)

{

perror(ERROR "shell : couldn't redirect, invalid input file ");

if (input\_file)

close(fd\_in);

free\_ptrs(arg\_count, arguments);

close(fd\_out);

return;

}

}

// Execute command based on the first argument

if (strcmp(arguments[0], "ls") == 0)

{

execute\_ls(arg\_count, arguments);

}

else if (strcmp("cd", arguments[0]) == 0)

{

execute\_cd(arg\_count, arguments);

}

else if (strcmp("echo", arguments[0]) == 0)

{

execute\_echo(arg\_count, arguments);

}

else if (strcmp("pwd", arguments[0]) == 0)

{

execute\_pwd();

}

else if (strcmp("history", arguments[0]) == 0)

{

history(arg\_count, arguments);

}

else if (strcmp("pinfo", arguments[0]) == 0)

{

execute\_pinfo(arg\_count, arguments);

}

else if (strcmp("fg", arguments[0]) == 0)

{

is\_fg = 1;

execute\_fg(arg\_count, arguments);

is\_fg = -1;

}

else if (strcmp("repeat", arguments[0]) == 0)

{

execute\_repeat(arg\_count, arguments);

}

else if (strcmp("bg", arguments[0]) == 0)

{

execute\_bg(arg\_count, arguments);

}

else if (strcmp("exit", arguments[0]) == 0)

{

// Handle exit command

write\_back\_history();

printf("\n\n\n");

fprintf(stderr, PROMPT "EVERY MEETING HAS A PARTING ;(\nSEE YOU AGAIN SOON !!!!!!!" RESET);

printf("\n\n\n");

exit(0);

}

else if (strcmp("replay", arguments[0]) == 0)

{

execute\_replay(arg\_count, arguments);

}

else if (strcmp("jobs", arguments[0]) == 0)

{

execute\_jobs(arg\_count, arguments);

}

else if (strcmp("sig", arguments[0]) == 0)

{

execute\_sig(arg\_count, arguments);

}

else

{

execute\_sys\_command(arg\_count, arguments);

}

// Restore standard input and output

if (in\_file)

{

int flag = dup2(stdin\_copy, STDIN\_FILENO);

if (flag < 0)

{

perror(ERROR);

fprintf(stderr, ERROR "EXITING ......\n");

free\_ptrs(arg\_count, arguments);

close(stdin\_copy);

close(stdout\_copy);

execute("exit");

}

close(fd\_in);

}

if (out\_file)

{

int flag = dup2(stdout\_copy, STDOUT\_FILENO);

if (flag < 0)

{

perror(ERROR);

fprintf(stderr, ERROR "EXITING ......\n");

free\_ptrs(arg\_count, arguments);

close(stdin\_copy);

close(stdout\_copy);

execute("exit");

}

close(fd\_out);

}

close(stdin\_copy);

close(stdout\_copy);

print\_RESET();

* check\_redirection(char comm): Checks if the input command contains redirection symbols '<' or '>'.
* free\_ptrs(int arg\_count, char arguments[]): Frees the memory allocated for command arguments.
* execute\_redirection(char comm): Executes the command with redirection specified in comm. Handles input and output redirection with or without append mode. Executes built-in commands or system commands based on the command arguments.

### Signals.c

#### **SIGINT\_HANDLER()**

void SIGINT\_HANDLER(int signum)

{

// Handle SIGINT signal (Ctrl+C)

pid\_t pid = getpid();

if (pid < 0)

{

perror(ERROR);

return;

}

else if (is\_fg == -1)

return;

else if (pid != SHELL\_PID)

{

// Terminate foreground child process

remove\_node(pid);

exit(EXIT\_FAILURE);

}

else

{

printf("\n");

fflush(stdout);

prompt(); // Print shell prompt after interrupt

}

}

#### **SIGCHILD\_HANDLER()**

void SIGCHILD\_HANDLER(int signum)

{

// Handle SIGCHLD signal (Child process status change)

int child\_status;

pid\_t child\_pid = waitpid(-1, &child\_status, WNOHANG | WUNTRACED);

if (child\_pid >= 0)

{

jobs proc = get\_data\_by\_pid(child\_pid);

if (proc != NULL)

{

if (WEXITSTATUS(child\_status) == 0 && WIFEXITED(child\_status))

{

fprintf(stderr, PROMPT "\nPROCESS %s WITH PID : %d EXITED NORMALLY \n" RESET, proc->proc\_name, child\_pid);

remove\_node(proc->proc\_pid);

}

else if (WIFSTOPPED(child\_status))

{

fprintf(stderr, ERROR "\nPROCESS %s WITH PID : %d STOPPED AFTER RECEIVING SIGNAL \n" RESET, proc->proc\_name, child\_pid);

}

else if (WIFSIGNALED(child\_status))

{

fprintf(stderr, ERROR "\nPROCESS %s WITH PID : %d EXITED AFTER RECEIVING SIGNAL \n" RESET, proc->proc\_name, child\_pid);

remove\_node(proc->proc\_pid);

}

else

{

fprintf(stderr, ERROR "\nPROCESS %s WITH PID : %d EXITED ABNORMALLY \n" RESET, proc->proc\_name, child\_pid);

remove\_node(proc->proc\_pid);

}

fflush(stderr);

prompt();

fflush(stdout);

}

}

}

#### **SIGTSTP\_HANDLER()**

void SIGTSTP\_HANDLER(int signum)

{

// Handle SIGTSTP signal (Ctrl+Z)

pid\_t pid = getpid();

if (pid < 0)

{

perror(ERROR);

return;

}

if (is\_fg == -1)

return;

if (pid != SHELL\_PID)

{

int flag = kill(pid, SIGTSTP);

if (flag < 0)

{

perror(ERROR);

}

}

else

printf("\n");

}

#### **sig()**

void sig(int job\_no, int sig\_num)

{

// Send signal to a specific job

jobs tmp = get\_data\_by\_id(job\_no);

if (tmp == NULL)

{

fprintf(stderr, ERROR "NO PROCESS EXISTS WITH GIVEN JOB ID!!!\n" RESET);

return;

}

print\_YELLOW();

printf("signaling proc %s : %d\n", tmp->proc\_name, tmp->proc\_pid);

int flag = kill(tmp->proc\_pid, sig\_num);

if (flag < 0)

{

perror(ERROR "sig ");

}

}

#### **execute\_sig()**

void execute\_sig(int arg\_count, char argument[])

{

// Execute the sig command

if (arg\_count != 3)

{

fprintf(stderr, ERROR "INVALID NUMBER OF ARGUMENTS, syntax - sig <job\_number> <signal\_number>\n" RESET);

return;

}

int job\_no = atoi(argument[1]);

if (job\_no <= 0)

{

fprintf(stderr, ERROR "ENTER A POSITIVE INTEGER FOR JOB NUMBER, syntax - sig <job\_number> <signal\_number>\n" RESET);

return;

}

int sig\_no = atoi(argument[2]);

if (sig\_no <= 0)

{

fprintf(stderr, ERROR "ENTER A POSITIVE INTEGER FOR SIGNAL NUMBER, syntax - sig <job\_number> <signal\_number>\n" RESET);

return;

}

sig(job\_no, sig\_no);

* This code defines signal handlers for SIGINT, SIGCHLD, and SIGTSTP, along with functions to handle specific signals and execute the sig command. It also includes error handling for various scenarios.

### System\_command.c

#### **execute\_sys\_command()**

void execute\_sys\_command(int arg\_count, char argument[])

{

bool is\_background\_proc = false;

if (arg\_count == 0)

{

perror(ERROR "Invalid Argument Called, give at least one command to execute\n");

return;

}

if (arg\_count >= 2 && strcmp(argument[arg\_count - 1], "&") == 0)

{

is\_background\_proc = true;

argument[arg\_count - 1] = NULL;

arg\_count--;

}

else if (arg\_count >= 1 && argument[arg\_count - 1][strlen(argument[arg\_count - 1]) - 1] == '&')

{

is\_background\_proc = true;

argument[arg\_count - 1][strlen(argument[arg\_count - 1]) - 1] = '\0';

argument[arg\_count] = NULL;

}

if (is\_background\_proc == false)

{

// Foreground process

argument[arg\_count] = NULL;

pid\_t proc\_fork = fork();

if (proc\_fork < 0)

{

perror(ERROR "Failed To Execute Command ");

return;

}

else if (proc\_fork == 0)

{

// Child process

signal(SIGINT, SIG\_DFL);

signal(SIGTSTP, SIG\_DFL);

setpgid(0, 0);

int exec\_flag = execvp(argument[0], argument);

if (exec\_flag < 0)

{

perror(ERROR "Failed To Execute Command ");

fprintf(stderr, RESET);

exit(EXIT\_FAILURE);

}

}

else

{

// Parent process

dup2(copy\_stdin\_fileno, 0);

dup2(copy\_stdout\_fileno, 1);

signal(SIGTTIN, SIG\_IGN);

signal(SIGTTOU, SIG\_IGN);

int st\_wait;

setpgid(proc\_fork, 0);

tcsetpgrp(STDIN\_FILENO, proc\_fork);

waitpid(proc\_fork, &st\_wait, WUNTRACED);

pid\_t pgid\_parent = getpgrp();

tcsetpgrp(STDIN\_FILENO, pgid\_parent);

signal(SIGTTIN, SIG\_DFL);

signal(SIGTTOU, SIG\_DFL);

if (WIFSTOPPED(st\_wait))

{

add\_node(argument[0], proc\_fork);

printf("Process %s with pid %d stopped .... \n", argument[0], proc\_fork);

}

}

}

else

{

// Background process

pid\_t proc\_fork = fork();

if (proc\_fork < 0)

{

perror(ERROR "Failed To Execute Command");

fprintf(stderr, RESET);

return;

}

else if (proc\_fork == 0)

{

// Child process

signal(SIGINT, SIG\_DFL);

signal(SIGTSTP, SIG\_DFL);

setpgid(0, 0);

int exec\_flag = execvp(argument[0], argument);

if (exec\_flag < 0)

{

perror(ERROR "Failed To Execute Command");

fprintf(stderr, RESET);

exit(EXIT\_FAILURE);

}

}

else

{

// Parent process

dup2(copy\_stdin\_fileno, 0);

dup2(copy\_stdout\_fileno, 1);

char command = (char )malloc(sizeof(char) name\_len);

strcpy(command, argument[0]);

for (int i = 1; i < arg\_count && argument[i] != NULL; i++)

{

strcat(command, " ");

strcat(command, argument[i]);

}

add\_node(command, proc\_fork);

free(command);

pid\_t pgid\_parent = getpgrp();

setpgid(proc\_fork, 0);

tcsetpgrp(STDIN\_FILENO, pgid\_parent);

}

}

* This code executes system commands, handling foreground and background processes appropriately. It sets up signal handling and process groups for job control.

### Tokenize.c

#### **getList()**

char getList(char input)

{

char list = (char )malloc(100 sizeof(char ));

num\_command = 0;

char temp = strtok(input, ";");

while (temp != NULL)

{

list[num\_command] = temp;

if (temp[0] != '\n')

num\_command++;

temp = strtok(NULL, ";");

}

replaceTabs(list);

RemoveExtraSpaces(list);

return list;

}

* This function tokenizes the input command based on semicolons (;) and stores each token in an array.
* It initializes an array of pointers to strings (char list) and uses strtok to split the input string into tokens.
* The tokens are stored in the list array, and the number of tokens is counted in the variable num\_command.
* It then calls two helper functions, replaceTabs and RemoveExtraSpaces, to clean up the tokens.
* Finally, it returns the array of tokens.

#### **replaceTabs()**

void replaceTabs(char list)

{

for (int i = 0; i < num\_command; i++)

{

for (int j = 0; j < strlen(list[i]); j++)

{

if (list[i][j] == '\t')

list[i][j] = ' ';

}

}

}

* This function replaces any tab characters (\t) in each token with spaces ( ).
* It iterates through each token in the list and checks for tab characters, replacing them with spaces.

#### **RemoveExtraSpaces()**

void RemoveExtraSpaces(char list)

{

for (int itr = 0; itr < num\_command; itr++)

{

char string = list[itr];

char cpyCommand[name\_len];

int j = 0;

while (string[j] == ' ')

j++;

bool prevspace = false;

int i = 0;

while (1)

{

if (string[j] != ' ')

{

cpyCommand[i] = string[j];

j++;

i++;

prevspace = false;

}

else if (string[j] == ' ' && prevspace == false)

{

cpyCommand[i] = string[j];

j++;

i++;

prevspace = true;

}

else

{

j++;

}

if (j == strlen(string))

break;

}

if (cpyCommand[i - 1] == ' ')

cpyCommand[i - 1] = '\0';

else

cpyCommand[i] = '\0';

// copy the string to original store placeholder

strcpy(list[itr], cpyCommand);

}

}

* This function removes extra spaces from each token in the list.
* It iterates through each token in the list and removes consecutive spaces, leaving only single spaces between words.
* The cleaned-up token is stored back in the original array.

## 4. System Calls Explanation

The project makes use of several system calls for process management:

* fork(): Create a new process by duplicating the calling process.
* exec(): Execute a file in the context of the calling process.
* wait(): Wait for a child process to terminate.
* kill(): Send a signal to a process or a group of processes.
* getpid(): Get the process ID of the calling process.
* getppid(): Get the parent process ID of the calling process.
* setpgid(): Set the process group ID of a specified process.

## 5. Parameters to Pass

### void execute\_bg(int arg\_count, char arguments[])

* Parameters:
* int arg\_count: The number of arguments.
* char arguments[]: An array of argument strings.
* Description: Resumes a background process.

### void execute\_cd(int arg\_count, char arguments[])

* Parameters:
* int arg\_count: The number of arguments.
* char arguments[]: An array of argument strings.
* Description: Changes the current directory.

### void execute\_echo(int arg\_count, char arguments[])

* Parameters:
* int arg\_count: The number of arguments.
* char arguments[]: An array of argument strings.
* Description: Prints the given arguments to the standard output.

### void execute(char comm)

* Parameters:
* char comm: The command string to execute.
* Description: Executes a command string.

### void execute\_fg(int arg\_count, char arguments[])

* Parameters:
* int arg\_count: The number of arguments.
* char arguments[]: An array of argument strings.
* Description: Resumes a foreground process.

### void trimLeading(char str)

* Parameters:
* char str: The string to trim.
* Description: Trims leading spaces from a string.

### void trimTrailing(char str)

* Parameters:
* char str: The string to trim.
* Description: Trims trailing spaces from a string.

### void history(int arg\_count, char arguments[])

* Parameters:
* int arg\_count: The number of arguments.
* char arguments[]: An array of argument strings.
* Description: Displays command history.

### void write\_back\_history()

* Parameters: None
* Description: Writes command history to a file.

### void execute\_jobs(int arg\_count, char arguments[])

* Parameters:
* int arg\_count: The number of arguments.
* char arguments[]: An array of argument strings.
* Description: Displays information about background jobs.

### void execute\_ls(int arg\_count, char arguments[])

* Parameters:
* int arg\_count: The number of arguments.
* char arguments[]: An array of argument strings.
* Description: Executes the ls command.

### void execute\_pinfo(int arg\_count, char arguments[])

* Parameters:
* int arg\_count: The number of arguments.
* char arguments[]: An array of argument strings.
* Description: Displays process information.

### void execute\_replay(int arg\_count, char arguments[])

* Parameters:
* int arg\_count: The number of arguments.
* char arguments[]: An array of argument strings.
* Description: Re-executes a specified command multiple times.

### void execute\_procmon()

* Parameters: None
* Description: This function doesn't take any parameters as inputs. It operates on the /proc directory to monitor running processes and their details.

### void init\_shell()

* Parameters: None
* Description: Initializes the shell environment, including clearing the screen, displaying welcome messages, and initializing various variables.

### void prompt()

* Parameters: None
* Description: Generates and prints the command prompt, including username, hostname, and current directory.

### void execute\_pwd()

* Parameters: None
* Description: Executes the pwd command to print the current working directory.

### int check\_redirection(char comm)

* Parameters:
* char comm: The command string to check for redirection symbols (< or >).
* Description: Checks if the given command contains input or output redirection symbols.

### void execute\_redirection(char comm)

* Parameters:
* char comm: The command string to execute with redirection.
* Description: Executes a command with input/output redirection based on the provided command string.

### void SIGINT\_HANDLER(int signum)

* Parameters:
* int signum: The signal number (SIGINT).
* Description: Handles the SIGINT signal (Ctrl+C) by either terminating the current process or printing the prompt.

### void SIGCHILD\_HANDLER(int signum)

* Parameters:
* int signum: The signal number (SIGCHLD).
* Description: Handles the SIGCHLD signal (child process status change) by printing process status messages.

### void SIGTSTP\_HANDLER(int signum)

* Parameters:
* int signum: The signal number (SIGTSTP).
* Description: Handles the SIGTSTP signal (Ctrl+Z) by suspending or resuming the current process.

### void sig(int job\_no, int sig\_num)

* Parameters:
* int job\_no: The job number.
* int sig\_num: The signal number.
* Description: Sends a signal to a specific job identified by its job number.

### void execute\_sig(int arg\_count, char argument[])

* Parameters:
* int arg\_count: The number of arguments.
* char argument[]: An array of argument strings.
* Description: Parses and executes the sig command to send a signal to a specific job.

### void execute\_sys\_command(int arg\_count, char argument[])

* Parameters:
* int arg\_count: The number of arguments.
* char argument[]: An array of argument strings.
* Description: Executes a system command with or without redirection and handles foreground or background execution.

### char getList(char input)

* Parameters:
* char input: The input command string.
* Description: Tokenizes the input command string based on semicolons and cleans up extra spaces.

### void replaceTabs(char list)

* Parameters:
* char list: An array of token strings.
* Description: Replaces tabs with spaces in each token string.

### void RemoveExtraSpaces(char list)

* Parameters:
* char list: An array of token strings.
* Description: Removes extra spaces from each token string.

### void execute\_help()

* Parameters: None
* Description: Displays help information about available shell commands.

### void execute\_find(int arg\_count, char arguments[])

* Parameters:
* int arg\_count: The number of arguments.
* char arguments[]: An array of argument strings.
* Description: Searches for files and directories.

### void execute\_pipe(int arg\_count, char arguments[])

* Parameters:
* int arg\_count: The number of arguments.
* char arguments[]: An array of argument strings.
* Description: Executes commands with pipes

### char read\_input()

* Parameters: None
* Description: Reads input from the user.
* Returns: A dynamically allocated string containing the user input.

## 6. Test Cases

### Test Case 1: Basic Input Processing

#### Description:

* Input a simple command without any special characters or options.

#### Expectation:

* The shell should correctly parse and execute the command.
* The output should match the expected behavior of the command.

#### Outcome:

* Test passed.

### Test Case 2: Redirection Handling

#### Description:

* Input a command with input/output redirection (`<`, `>`, `>>`).

#### Expectation:

* Verify that input/output redirection works as intended.
* Ensure that the command produces the expected output after redirection.

#### Outcome:

* Test passed.

### Test Case 3: Command Execution

#### Description:

* Input various system commands and built-in shell commands.

#### Expectation:

* Each command should be executed correctly by the shell.
* The shell behavior should align with the expected behavior of the command.

#### Outcome:

* Test passed.

### Test Case 4: Error Handling

#### Description:

* Input commands with incorrect syntax or nonexistent commands.

#### Expectation:

* The shell should provide clear and informative error messages for invalid commands.
* It should gracefully handle errors without crashing or disrupting the shell session.

#### Outcome:

* Test passed.

### Test Case 5: Background Process Execution

#### Description:

* Input commands with the `&` symbol to run them as background processes.

#### Expectation:

* Background processes should execute concurrently without blocking the shell.
* The shell prompt should return immediately after initiating a background process.

#### Outcome:

* Test passed.

### Test Case 6: Signal Handling

#### Description:

* Send various signals (`SIGINT`, `SIGTSTP`, etc.) to running processes.

#### Expectation:

* The shell should correctly intercept and handle signals according to predefined signal handlers.
* Processes should respond appropriately to signals without crashing or hanging.

#### Outcome:

* Test passed.

## 7. References

* *https://github.com/tokenrove/build-your-own-shell*
* *https://www.gnu.org/software/libc/manual/html\_node/Implementing-a-Shell.html*