`

Mohammad Hadi Sormeyli(99463143)

OS project

Introduction

This document provides an explanation of the code written in C language. The code is a simple shell program that allows users to execute commands, manage background processes, and perform other basic shell functionalities. The program utilizes threads and synchronization mechanisms to handle background processes and their statuses.

Code Documentation

The code provided is an implementation of a simple shell program called "IMCSH" (Interactive Multi-Command Shell). It allows users to execute commands, manage background processes, and perform various operations. Let's go through the different parts of the code and understand their functionalities.

Header Files

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <pthread.h>
```

- `stdio.h` is included for standard input/output operations.
- `stdlib.h` is included for memory allocation and process control functions.
- `string.h` is included for string manipulation functions.
- `unistd.h` is included for POSIX operating system API functions such as fork and exec.
- `sys/types.h` is included for definitions of data types used in system calls.
- `sys/wait.h` is included for process wait functions.
- `pthread.h` is included for thread-related functions and synchronization mechanisms.

Macros

```
#define AUTHORS "HADI"
#define MAX_COMMAND_LENGTH 1024
#define MAX_ARGS 64
#define MAX_BACKGROUND_PROCESSES 10
```

These macros define some constants used in the program:

- `AUTHORS`: Represents the name of the authors (currently set as "HADI").
- `MAX_COMMAND_LENGTH`: Specifies the maximum length of a command string.
- `MAX_ARGS`: Defines the maximum number of arguments a command can have.
- `MAX_BACKGROUND_PROCESSES`: Sets the maximum number of background processes that can be running simultaneously.

Data Structures

```
struct background_process {
    pid_t pid;
    char command[MAX_COMMAND_LENGTH];
};

struct background_process background_processes[MAX_BACKGROUND_PROCESSES];
int num_background_processes = 0;
```

The code defines a structure `background_process` to store information about a background process. It contains two fields: `pid` (process ID) and `command` (the command associated with the process). An array `background_processes` of type `background_process` is used to store multiple background processes. The variable `num_background_processes` keeps track of the number of background processes currently running.

Global Variables

```
pthread_mutex_t mutex;
```

A global variable `mutex` of type `pthread_mutex_t` is declared to synchronize access to shared resources when managing background processes using threads.

Function Declarations

```
void* check_background_process(void* arg);
void print_background_processes();
void add_background_process(pid_t pid, char *command);
void remove_background_process(int index);
int check_process_status(int p_id);
char* get_output_file(char* command);
int exec(char *command, int background);
```

```
void global_usage(char* command);
```

These are function declarations for the various operations performed in the program. We'll explain each of these functions in detail later.

Background Process Management Functions

There are several functions related to managing background processes:

Background Process Checker Thread

```
void* check_background_process(void* arg) {
    pthread_mutex_lock(&mutex);

    while(1) {
        for(int i = 0;i < num_background_processes;i++) {
            int status = check_process_status(background_processes[i].pid);

            if(status == 0) {
                printf("\nChild process with PID %d exited with status: %d\n",
background_processes[i].pid, WEXITSTATUS(status));
                remove_background_process(i);
            }
        }
    }

    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

This function is a thread routine that continuously checks the status of background processes. It acquires the lock on the `mutex` variable to ensure mutual exclusion while accessing shared resources. Inside an infinite loop, it iterates over all the background processes and checks their status using the `check_process_status()` function. If a background process has exited (status equals 0), it removes it from the list of background processes using the `remove_background_process()` function. The thread routine releases the lock on the `mutex` variable before exiting.

Printing Background Processes

The `print_background_processes` function prints information about the current background processes:

```
void print_background_processes() {
    printf("Background Processes:\n");
    for (int i = 0; i < num_background_processes; i++) {
        printf("[%d] %d %s\n", i + 1, background_processes[i].pid,
background_processes[i].command);
    }
}
```

It iterates over the `background_processes` array and prints the process index, PID, and the command associated with each background process.

Adding Background Process

The `add_background_process` function adds a background process to the `background_processes` array:

```
void add_background_process(pid_t pid, char *command) {
    if (num_background_processes < MAX_BACKGROUND_PROCESSES) {
        background_processes[num_background_processes].pid = pid;
        strncpy(background_processes[num_background_processes].command, command,
MAX_COMMAND_LENGTH);
        num_background_processes++;
    } else {
        printf("Maximum background processes reached.\n");
    }
}
```

It checks if the maximum number of background processes has been reached. If not, it assigns the `pid` and `command` values to the next available slot in the `background_processes` array and increments `num_background_processes`.

Removing Background Process

The `remove_background_process` function removes a background process from the `background_processes` array:

```
void remove_background_process(int index) {
    for (int i = index; i < num_background_processes - 1; i++) {
        background_processes[i] = background_processes[i + 1];
    }
    num_background_processes--;
}
```

It shifts the elements in the `background_processes` array starting from the specified `index` to the left, effectively removing the process at that index. The `num_background_processes` variable is decremented to reflect the removal.

Checking Process Status

The `check_process_status` function checks the status of a process using the `waitpid` system call:

```
int check_process_status(int p_id) {
    int status;
    waitpid(p_id, &status, 0);
    if (WIFEXITED(status)) {
        return 0;
    } else if (WIFSIGNALED(status)) {
        return 1;
    }
    return 0;
}
```

It waits for the process with the given `p_id` to change state and stores the exit status or termination signal in the `status` variable. If the process exited normally (`WIFEXITED`), it returns 0. If the process was terminated by a signal (`WIFSIGNALED`), it returns 1. Otherwise, it returns 0.

Redirecting Output

The `get_output_file` function extracts the output file name from a command:

```
char* get_output_file(char* command) {
    char *output_file = NULL;
    char *output_symbol = strchr(command, '>');
    if (output_symbol != NULL) {
        output_symbol++;
        if (*output_symbol == ' ') {
            output_symbol++;
        }
        output_file = strtok(output_symbol, " \n");

        command = strtok(command, ">");
        return output_file;
    }
    return NULL;
}
```

It searches for the `'>'` symbol in the command string and extracts the output file name. The output file name is returned if found; otherwise, it returns NULL.

Executing a Command

The `exec` function parses and executes a command:

```
int exec(char *command, int background) {
    char* output_file = get_output_file(command);

    char *token = strtok(command, " \n");
    char *args[MAX_ARGS];
    int arg_count = 0;

    while (token != NULL) {
        args[arg_count++] = token;
        token = strtok(NULL, " \n");
    }

    args[arg_count] = NULL;

    pid_t child_pid = fork();

    if(child_pid < 0) {
        perror("Fork failed");
        return -1;
    }

    if(child_pid == 0) {
        if (output_file != NULL) {
            FILE *file = freopen(output_file, "a", stdout);
            if (file == NULL) {
                perror("Failed to redirect output");
                exit(EXIT_FAILURE);
            }
            printf("%s", stdout);
        } else if(background) {
            freopen("/dev/null", "w", stdout);
            freopen("/dev/null", "w", stderr);
        }

        execvp(args[0], args) ;
        perror("Command execution failed");
        exit(0);
    }

    if (background) {
        printf("Background process started with PID %d\n", child_pid);
        add_background_process(child_pid, command);
    } else {
        int status = check_process_status(child_pid);
```

```
        if (status == 0) {
            printf("Child process with PID %d exited with status: %d\n", child_pid,
WEXITSTATUS(status));
        } else if (status == 1) {
            printf("Child process with PID %d was terminated by signal: %d\n", child_pid,
WTERMSIG(status));
        }
    }

    return 0;
}
```

The function first calls `get_output_file` to determine if output redirection is required. It then tokenizes the command string using whitespace and newline characters (`" \n"`). The tokens are stored in the `args` array, and the number of arguments is tracked by `arg_count`. A `NULL` terminator is added at the end of the `args` array.
A child process is created using `fork()`. If the creation of the child process fails, an error message is printed. In the child process, it checks for output redirection and redirects `stdout` to the specified output file using `freopen`. If no output redirection is needed but the command is executed in the background, both `stdout` and `stderr` are redirected to `/dev/null`.
The `execvp` function is called to replace the child process's image with the specified command.

Global usage

The `global_usage()` function is responsible for displaying the global usage information of the shell program. It takes a command string as input and provides functionality to redirect the output to a file or print it to the console.

```
char des[100];
snprintf(des, sizeof(des), "IMCSH Version 1.1 created by <%s>\n", AUTHORS);
```

A character array `des` is created to store the description. The `snprintf()` function is used to format the description string with the version number (1.1) and the author's name specified in the `AUTHORS` macro.

```
char* output_file = get_output_file(command);

if (output_file != NULL) {
    FILE *file = fopen(output_file, "a");
    if (file == NULL) {
        perror("Failed to redirect output");
        exit(EXIT_FAILURE);
    }

    fputs(des, file);
    fclose(file);
} else {
```

```
    printf("%s", des);
}
```

The function first checks if there is an output file specified in the command using the `get_output_file()` function. If an output file is found, it opens the file in append mode (`"a"`) and checks if the file was opened successfully. If the file opening fails, an error message is printed, and the program exits.
If the file opening is successful, the description string `des` is written to the file using the `fputs()` function. Finally, the file is closed.
If no output file is specified in the command, the description string is printed to the console using `printf()`.

Main

The `main()` function is the entry point of the shell program. It initializes a background process checking thread, handles user input, and executes various commands based on the user's actions.

```
int main()
```

Functionality:

The `main()` function performs the following steps:

1. Initializes a background process checking thread:
    - Creates a pthread object `thread`.
    - Initializes a mutex `mutex` using `pthread_mutex_init()`.
    - Calls `pthread_create()` to create a new thread that will execute the `check_background_process()` function.
    - Locks and immediately unlocks the mutex using `pthread_mutex_lock()` and `pthread_mutex_unlock()` respectively. This ensures that the `check_background_process()` function executed by the thread has exclusive access to the shared resources protected by the mutex.
2. Declares variables for command input and program control:
    - Declares a character array `command` to store user command input.
    - Declares an integer variable `running` initialized to `1`, which controls the main program loop.
3. Enters the main program loop:
    - Prints a prompt indicating the name of the author (`AUTHORS`) to the console using `printf()`.
    - Waits for user input by reading a line of text into the `command` array using `fgets()`.

- Removes the trailing newline character from the `command` string using `strcspn()`.
- Checks the user's command and performs the corresponding action.

4. Command handling:
    - If the command is "quit", it checks if there are any background processes running. If there are, it prompts the user for confirmation before terminating the program. If the user confirms, it sets `running` to `0` to exit the loop and sends a termination signal (`SIGTERM`) to each background process using `kill()`.
    - If the command is "globalusage", it calls the `global_usage()` function, passing the `command` string as an argument. This displays the global usage information for the shell program.
    - If the command starts with "exec", it extracts the command to be executed from the input string and checks if it should run in the background. If the last character of the command is '&', it sets the `background` flag to `1` and removes the '&' character from the command string. It then calls the `exec()` function, passing the extracted command and the `background` flag as arguments, to execute the command.
    - If none of the above conditions are met, it prints an "Unrecognized command" message to the console.
5. Continues until the user chooses to quit:
    - The main program loop continues until the `running` variable is set to `0`, indicating that the user has chosen to quit.
6. Returns from the `main()` function:
    - The `main()` function returns `0`, indicating successful execution of the program.

The `main()` function coordinates the execution of various commands based on user input. It initializes the necessary resources, handles user commands, and controls the termination of the program.