Team Member : Manish Niure
Prem Oli

Approach to create a shell based on the conditions assigned in Codio to execute commands from the command line.

Task 1: Implementing Shell Prompt with Built-in Commands

Changing Directories

"cd" Command To switch between directories, we implemented the "cd" command. After checking that the first argument matches "cd", the in-built `chdir` function shifts the current directory to the provided path.

```
if (strcmp(arguments[0],"cd")==0){
    chdir(arguments[1]);
}
```

Displaying Current Directory

"pwd" Command The "pwd" command shows the present working directory. If the first argument is "pwd", we utilize the `getcwd` function to achieve this.

```
else if (strcmp(arguments[0],"pwd")==0){
    printf("%s\n",getcwd(wd,sizeof(wd)));
}
```

Displaying Messages & Environment Variables

"echo" Command The "echo" command can either print a direct message or fetch the value of an environment variable (when preceded by a "$" sign). We use the `getenv` function to retrieve environment variables.

```
else if (strcmp(arguments[0],"echo")==0){
    i = 1;
    while(arguments[i] != NULL){
        if (arguments[i][0] == '$'){
            printf("%s ", getenv(arguments[i]+1));
        } else {
            printf("%s ", arguments[i]);
        }
        i++;
    }
}
```

Setting Environment Variables

"setenv" Command For establishing new environment variables, we use the "setenv" command. After parsing the argument to extract the key and value, we employ the `setenv` function.

```
else if (strcmp(arguments[0],"setenv")==0) {
    char* temp_val[2];
    temp_val[0] = strtok(arguments[1], "=");
    i=0;
    while(temp_val[i] != NULL || i == 2) {
        i++;
        temp_val[i] = strtok(NULL, "=");
    }
    setenv(temp_val[0], temp_val[1], 1);
}
```

Exiting the Shell

"exit" Command To gracefully exit the shell, the "exit" command utilizes the inbuilt `exit` function.

```
else if (strcmp(arguments[0],"exit")==0){
    exit(0);
}
```

Displaying Environment Variables

"env" Command The "env" command exhibits all environment variables. If a specific variable is given as an argument, it fetches its value; otherwise, it lists all of them.

```
else if (strcmp(arguments[0],"env")==0){
    if (arguments[1] != NULL){
        printf("%s\n",getenv(arguments[1]));
    } else {
        char** env= environ;
        for (; *env; env++)
            printf("%s\n", *env);
    }
}
```

Task 2: Executing Processes

We utilized the **execute_command** function to process the provided arguments, allowing the input to be run as if they were native shell commands. For instance, when given the argument "ls -l", it operates just as the integrated shell command would.

```
int pid = fork();
if (pid == 0){
   signal(SIGINT, SIG_DFL);
   execute_command(arguments);
   exit(0);
}
```

Task 3: Adding background Process

We introduced a boolean variable to monitor the addition of background tasks. If the command line concludes with an "&", this boolean variable is set to true.

```
char* last_arg = arguments[i-1];
bool background_process = false;
if (strcmp(last_arg,"&")==0){

   background_process = true;
   arguments[i-1] = NULL;}
```

Task 4: Signal Handling

We devised a function called signal_handler to manage the Ctrl+C signal, allowing the termination of a foreground process. This is beneficial as it stops the shell itself from exiting when Ctrl+C is pressed.

```
// signal handling
void signal_handler(int signum)
{
   if (command_p != -1){
       kill(command_p, SIGINT);
   }
}
```

Task 5: Killing off long running processes

We designed a function called terminate_process, which stands for terminating the process. This function ends the foreground process if it hasn't completed after a duration of 10 seconds.

```
void terminate_process(int time, int pid) {
   sleep(time);
   printf("Time out foreground process.\n");
   kill(pid, SIGINT);
}
```

Extra Credit

The **>** symbol was identified to manage output redirection. When encountered, the subsequent argument was treated as the filename, and the command's output was directed to overwrite or create this file.

The **<** symbol facilitated input redirection. Upon its detection, the following argument specified the file from which input should be read for the command.

The | symbol was employed for command piping. Commands on either side of this symbol were processed such that the output of the first command became the input for the subsequent one.

```
    if (fd < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    dup2(fd, STDOUT_FILENO);
    close(fd);

    // Input redirection
} else if (strcmp(arguments[i], "<") == 0) {
    arguments[i] = NULL;
    file = arguments[++i];
    fd = open(file, O_RDONLY);
    if (fd < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    dup2(fd, STDIN_FILENO);
    close(fd);

    // Command piping
} else if (strcmp(arguments[i], "|") == 0) {
    if (command_token_count == 0) {
        printf("Invalid pipe command\n");
        return;
    }
    args_by_pipe[command_count][command_token_count] = NULL;
    command_count++;
    command_token_count = 0;
```