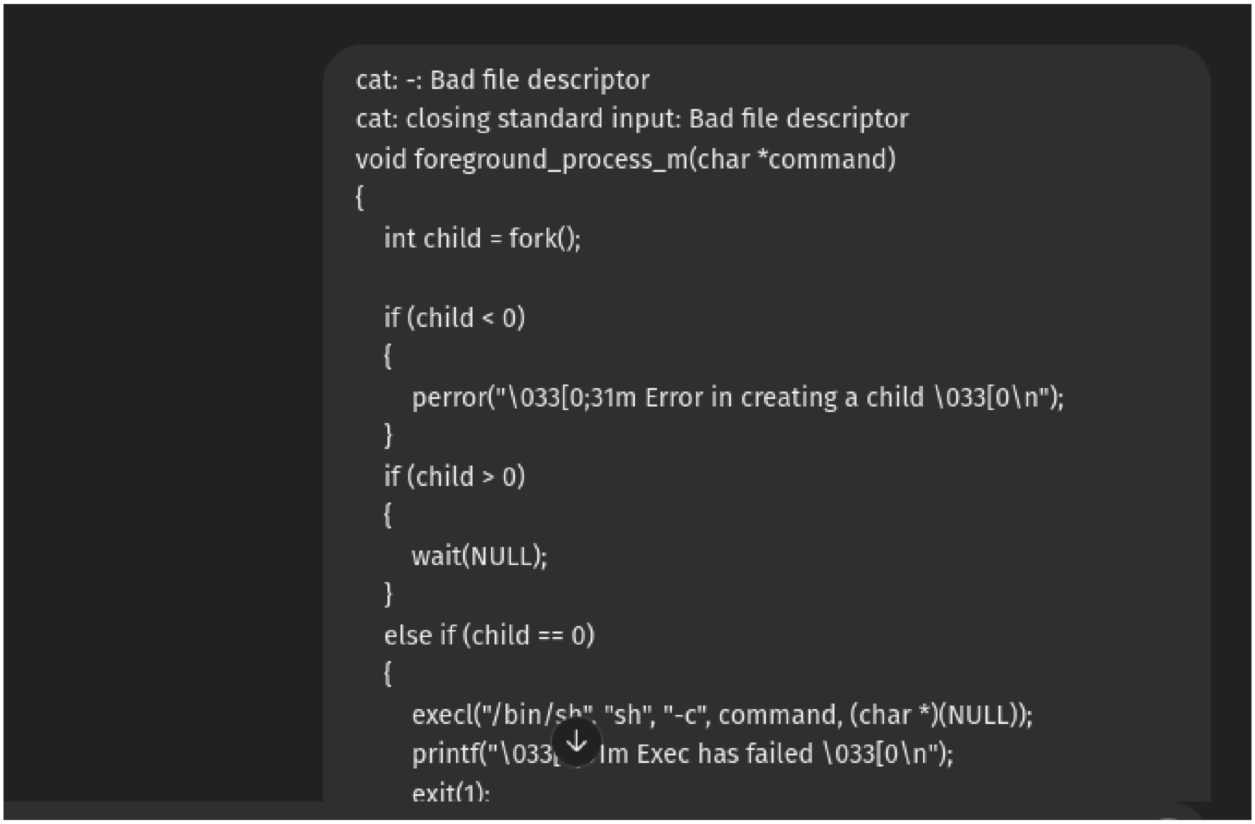
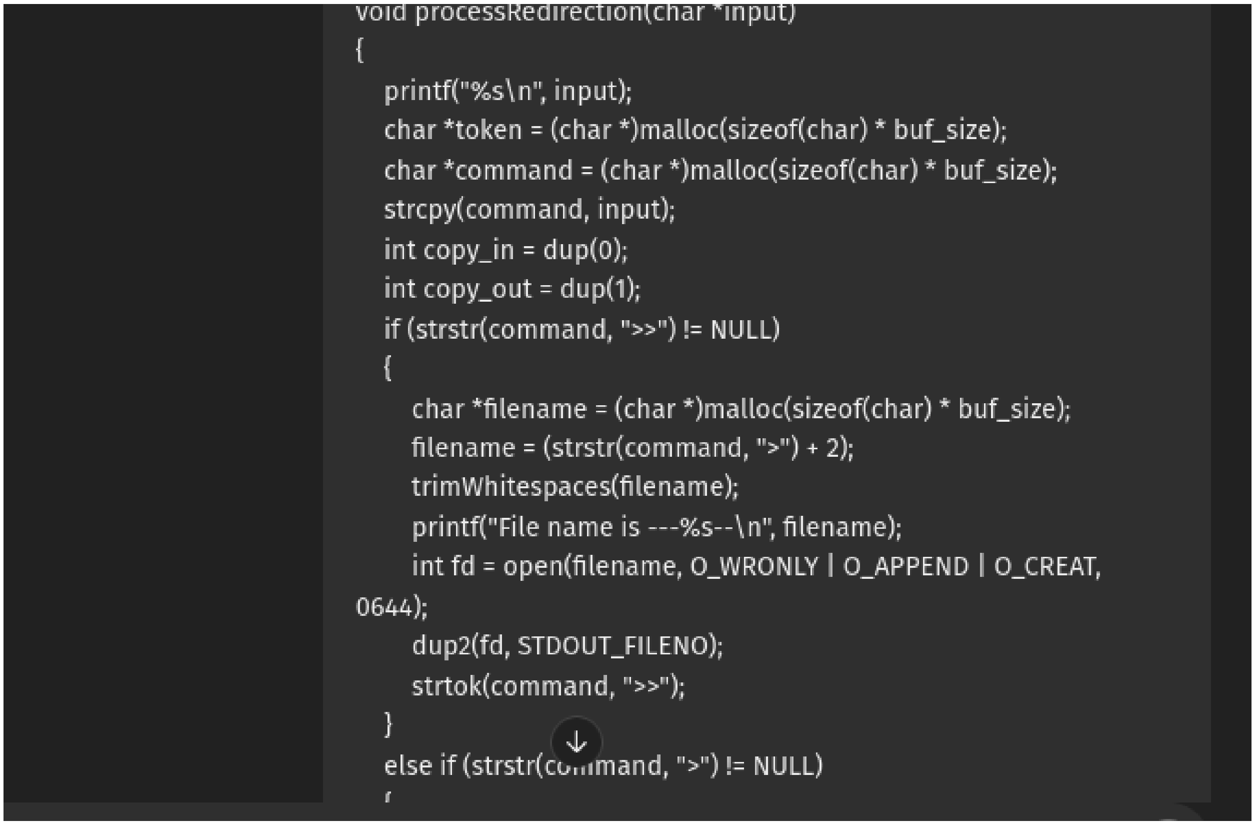
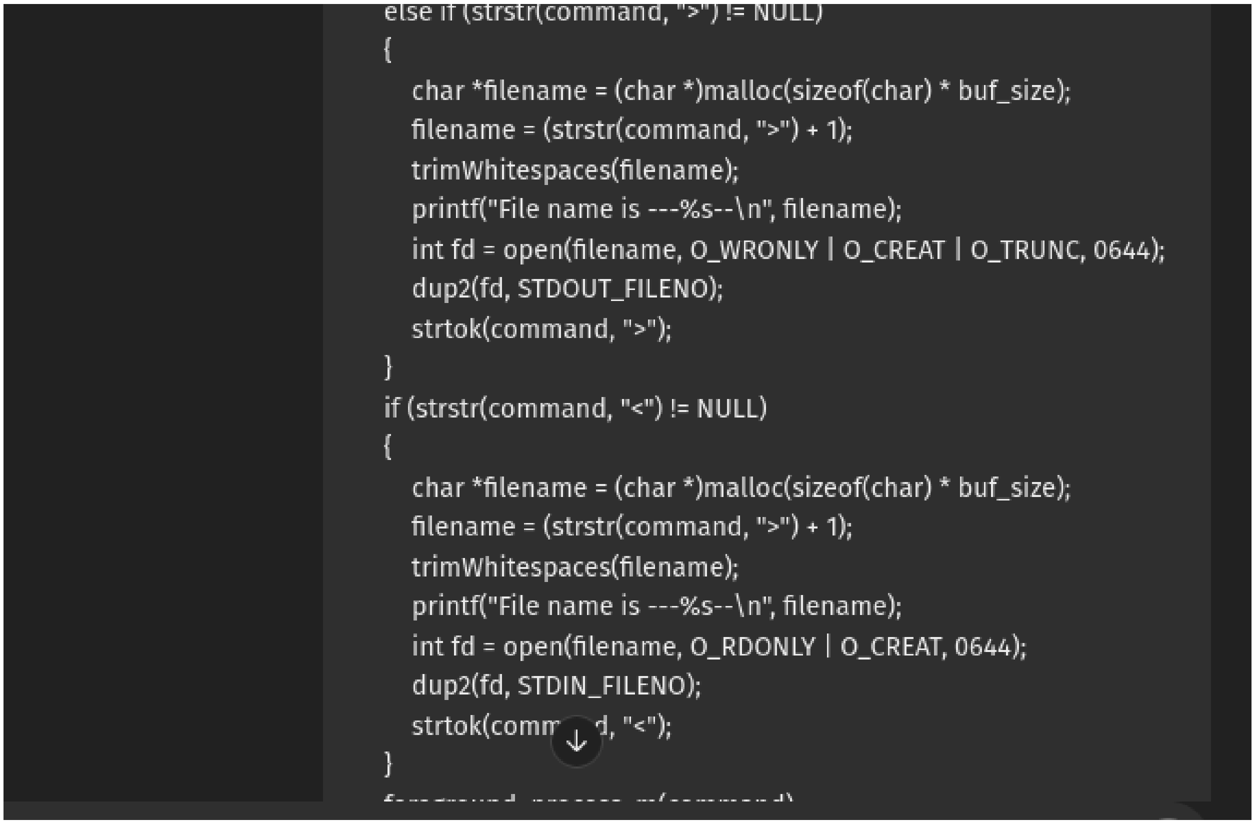


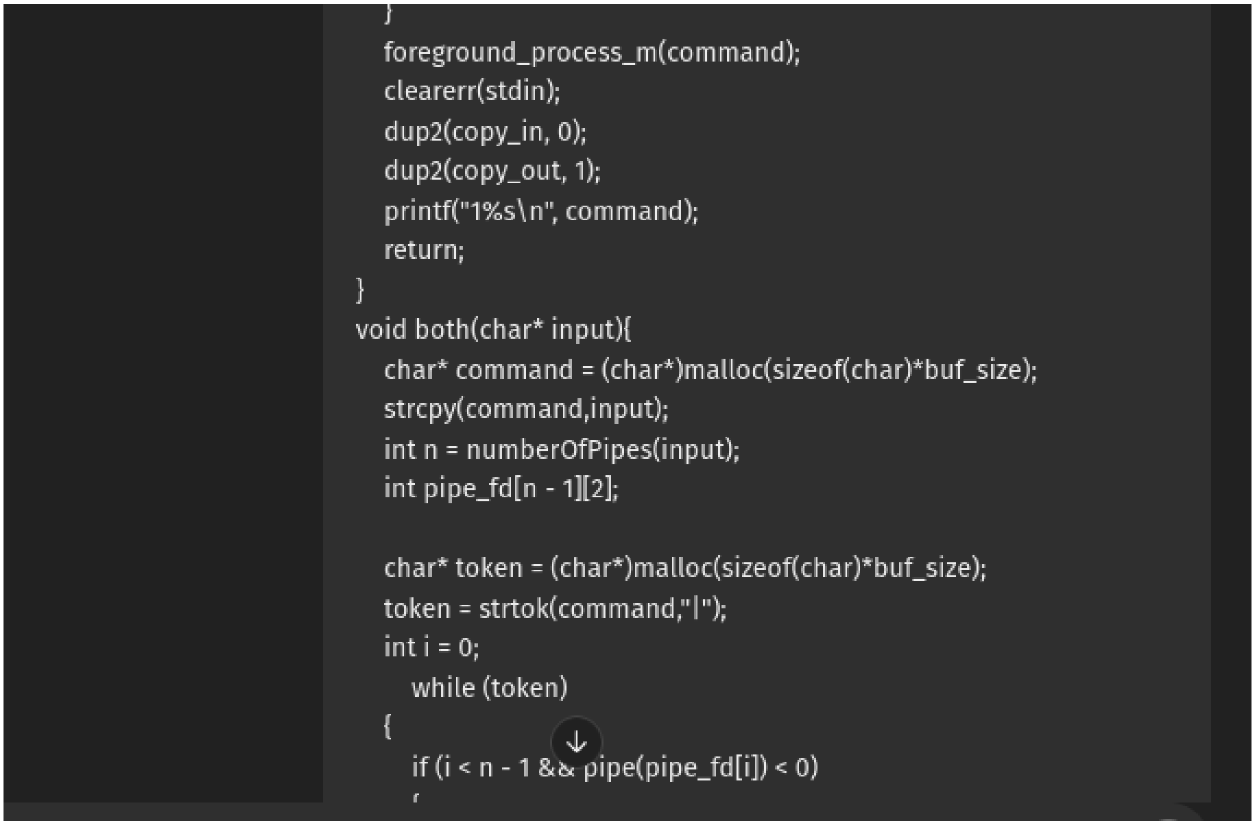
```
cat: -: Bad file descriptor
cat: closing standard input: Bad file descriptor
void foreground_process_m(char *command)
{
    int child = fork();

    if (child < 0)
    {
        perror("\033[0;31m Error in creating a child \033[0\n");
    }
    if (child > 0)
    {
        wait(NULL);
    }
    else if (child == 0)
    {
        execl("/bin/sh", "sh", "-c", command, (char *)(NULL));
        printf("\033[[illegible]1m Exec has failed \033[0\n");
        exit(1);
```



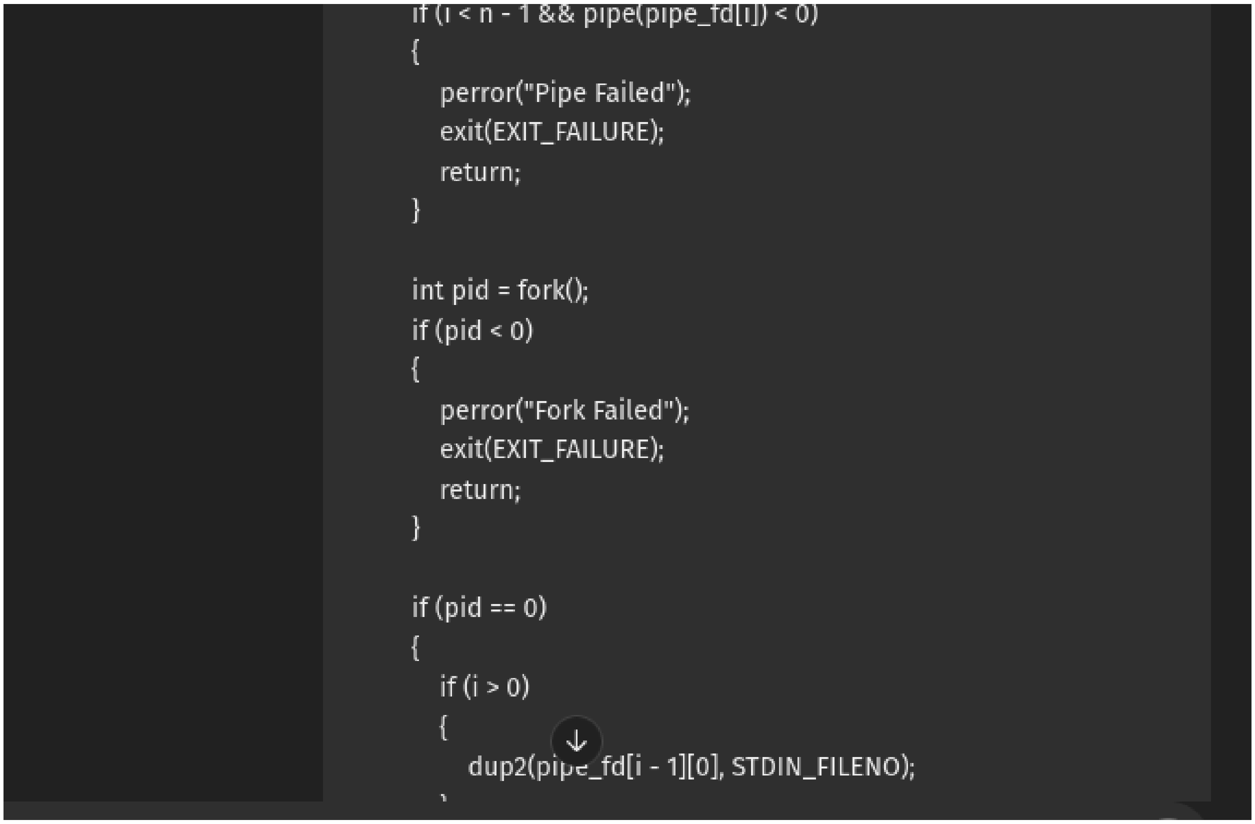
```
void processRedirection(char *input)
{
    printf("%s\n", input);
    char *token = (char *)malloc(sizeof(char) * buf_size);
    char *command = (char *)malloc(sizeof(char) * buf_size);
    strcpy(command, input);
    int copy_in = dup(0);
    int copy_out = dup(1);
    if (strstr(command, ">>") != NULL)
    {
        char *filename = (char *)malloc(sizeof(char) * buf_size);
        filename = (strstr(command, ">") + 2);
        trimWhitespaces(filename);
        printf("File name is ---%s--\n", filename);
        int fd = open(filename, O_WRONLY | O_APPEND | O_CREAT,
0644);
        dup2(fd, STDOUT_FILENO);
        strtok(command, ">>");
    }
    else if (strstr(command, ">") != NULL)
    {
```



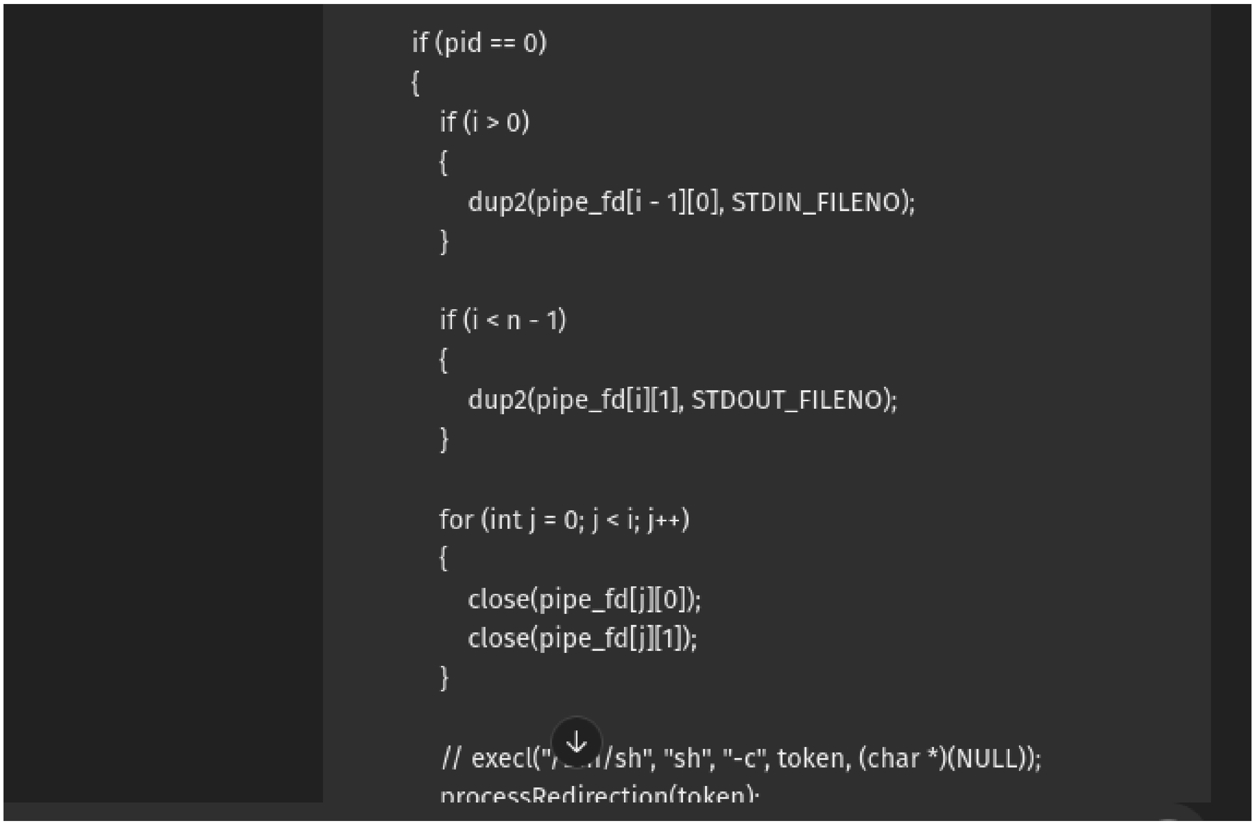
```
    else if (strstr(command, ">") != NULL)
    {
        char *filename = (char *)malloc(sizeof(char) * buf_size);
        filename = (strstr(command, ">") + 1);
        trimWhitespaces(filename);
        printf("File name is ---%s--\n", filename);
        int fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
        dup2(fd, STDOUT_FILENO);
        strtok(command, ">");
    }
    if (strstr(command, "<") != NULL)
    {
        char *filename = (char *)malloc(sizeof(char) * buf_size);
        filename = (strstr(command, ">") + 1);
        trimWhitespaces(filename);
        printf("File name is ---%s--\n", filename);
        int fd = open(filename, O_RDONLY | O_CREAT, 0644);
        dup2(fd, STDIN_FILENO);
        strtok(comm[illegible]d, "<");
    }
```



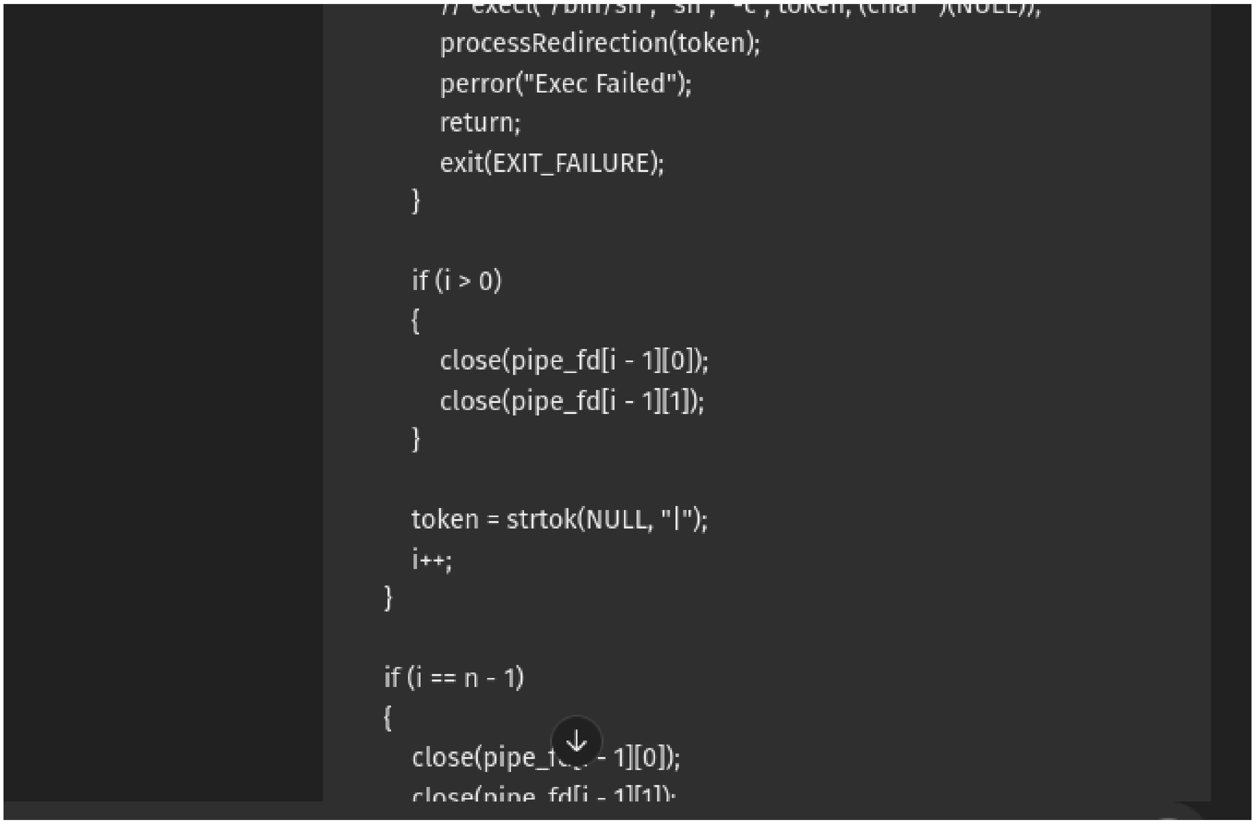
```
    }
    foreground_process_m(command);
    clearerr(stdin);
    dup2(copy_in, 0);
    dup2(copy_out, 1);
    printf("1%s\n", command);
    return;
}
void both(char* input){
    char* command = (char*)malloc(sizeof(char)*buf_size);
    strcpy(command,input);
    int n = numberOfPipes(input);
    int pipe_fd[n - 1][2];

    char* token = (char*)malloc(sizeof(char)*buf_size);
    token = strtok(command,"|");
    int i = 0;
        while (token)
    {
        if (i < n - 1 && pipe(pipe_fd[i]) < 0)
```



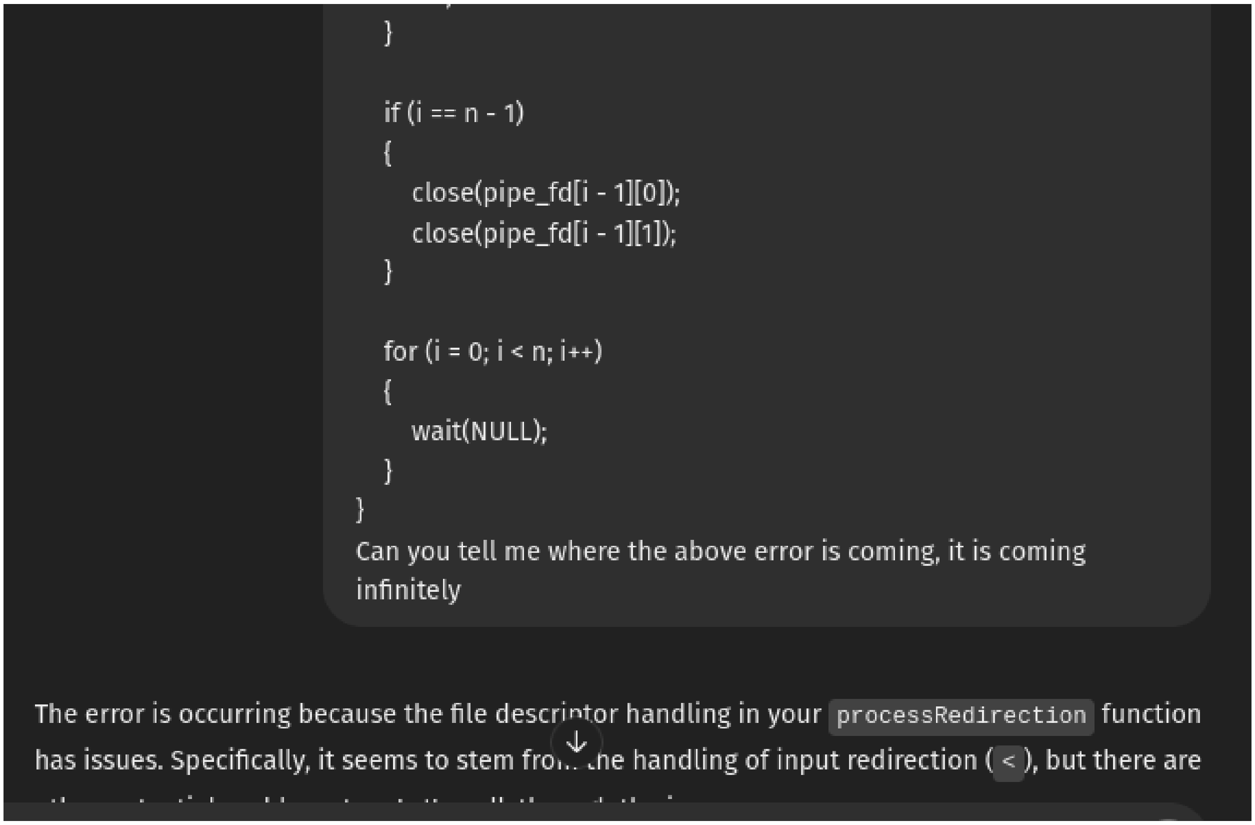
```
if (i < n - 1 && pipe(pipe_fd[i]) < 0)
{
    perror("Pipe Failed");
    exit(EXIT_FAILURE);
    return;
}

int pid = fork();
if (pid < 0)
{
    perror("Fork Failed");
    exit(EXIT_FAILURE);
    return;
}

if (pid == 0)
{
    if (i > 0)
    {
        dup2(pipe_fd[i - 1][0], STDIN_FILENO);
```



```
if (pid == 0)
{
    if (i > 0)
    {
        dup2(pipe_fd[i - 1][0], STDIN_FILENO);
    }

    if (i < n - 1)
    {
        dup2(pipe_fd[i][1], STDOUT_FILENO);
    }

    for (int j = 0; j < i; j++)
    {
        close(pipe_fd[j][0]);
        close(pipe_fd[j][1]);
    }

    // execl("/bin/sh", "sh", "-c", token, (char *)(NULL));
    processRedirection(token);
```



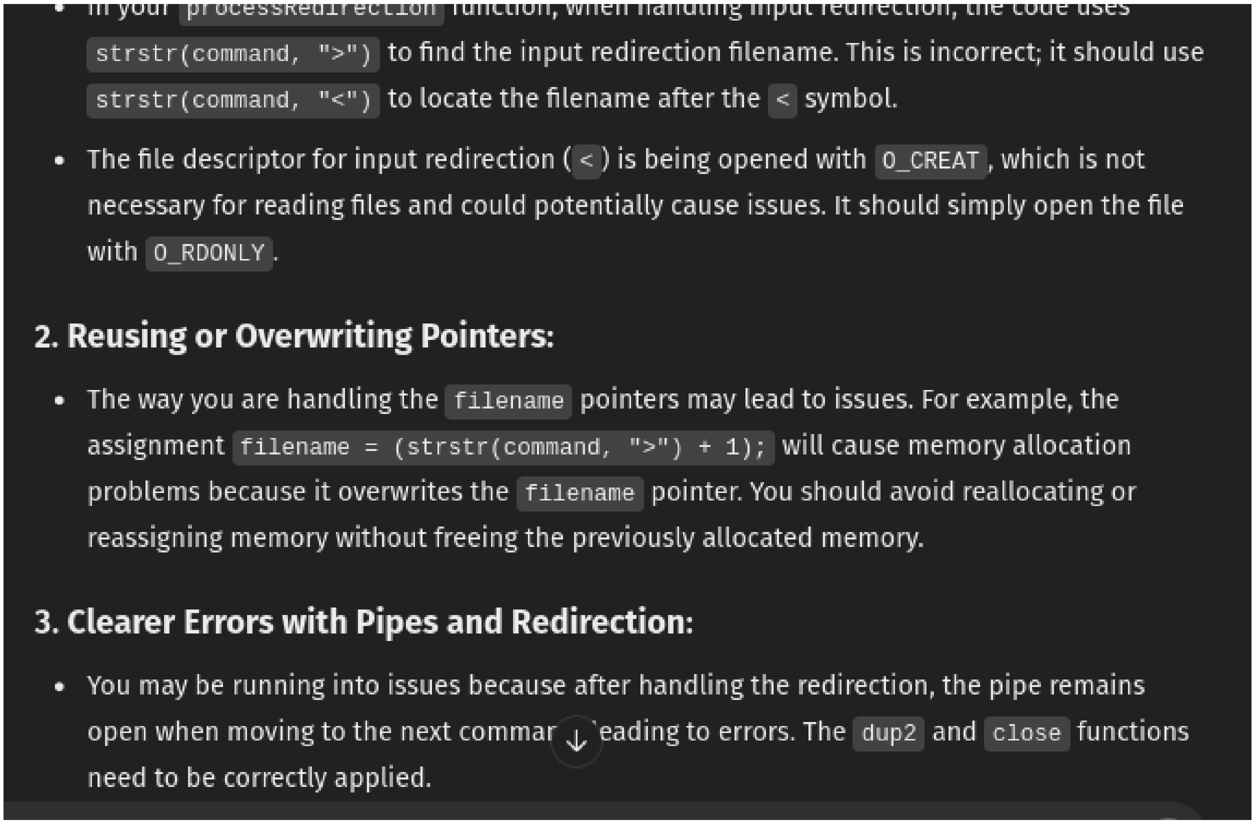
```
        // execl("/bin/sh", "sh", "-c", token, (char *)(NULL));
        processRedirection(token);
        perror("Exec Failed");
        return;
        exit(EXIT_FAILURE);
    }

    if (i > 0)
    {
        close(pipe_fd[i - 1][0]);
        close(pipe_fd[i - 1][1]);
    }

    token = strtok(NULL, "|");
    i++;
}

if (i == n - 1)
{
    close(pipe_fd[i - 1][0]);
    close(pipe_fd[i - 1][1]);
```



```
    }

    if (i == n - 1)
    {
        close(pipe_fd[i - 1][0]);
        close(pipe_fd[i - 1][1]);
    }

    for (i = 0; i < n; i++)
    {
        wait(NULL);
    }
}
```

Can you tell me where the above error is coming, it is coming infinitely

The error is occurring because the file descriptor handling in your `processRedirection` function has issues. Specifically, it seems to stem from the handling of input redirection (`<`), but there are



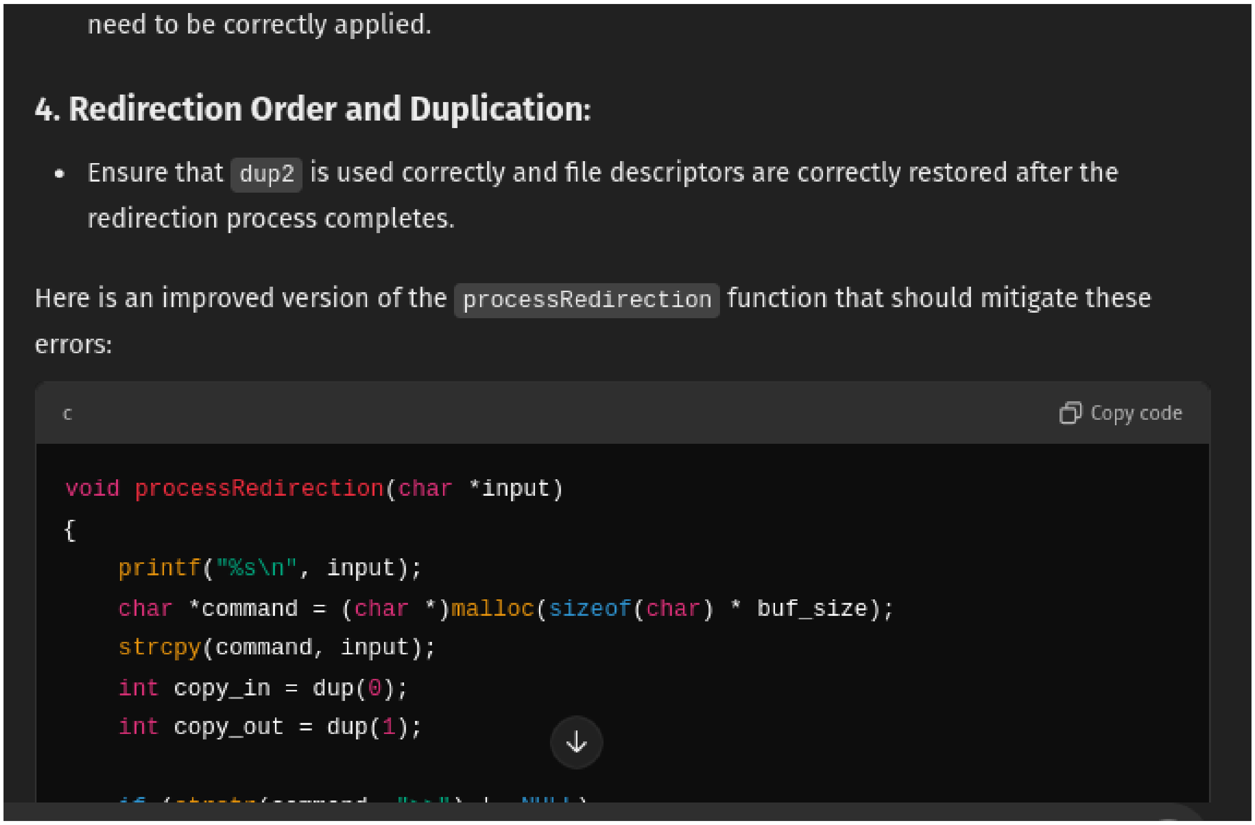
- In your `processRedirection` function, when handling input redirection, the code uses `strstr(command, ">")` to find the input redirection filename. This is incorrect; it should use `strstr(command, "<")` to locate the filename after the `<` symbol.
- The file descriptor for input redirection (`<`) is being opened with `O_CREAT`, which is not necessary for reading files and could potentially cause issues. It should simply open the file with `O_RDONLY`.

2. Reusing or Overwriting Pointers:

- The way you are handling the `filename` pointers may lead to issues. For example, the assignment `filename = (strstr(command, ">") + 1);` will cause memory allocation problems because it overwrites the `filename` pointer. You should avoid reallocating or reassigning memory without freeing the previously allocated memory.

3. Clearer Errors with Pipes and Redirection:

- You may be running into issues because after handling the redirection, the pipe remains open when moving to the next command, leading to errors. The `dup2` and `close` functions need to be correctly applied.



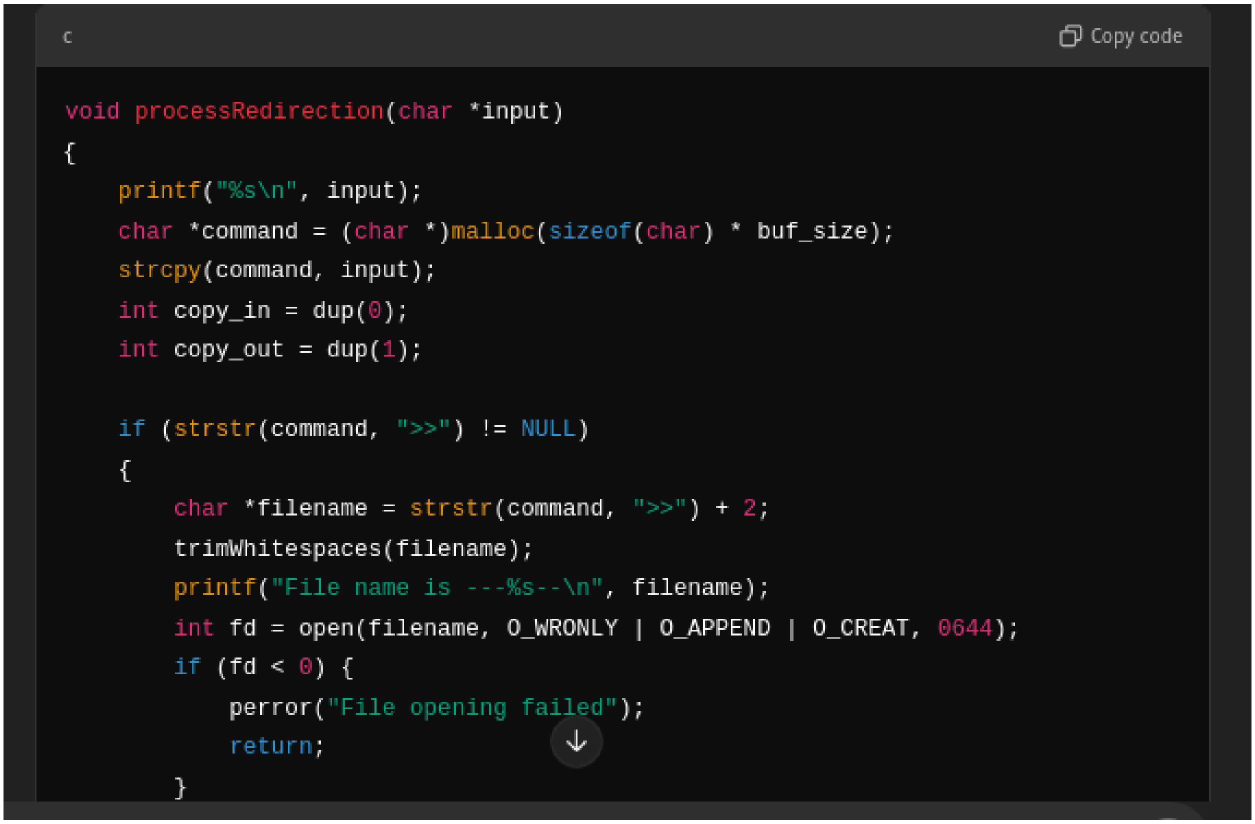
need to be correctly applied.

4. Redirection Order and Duplication:

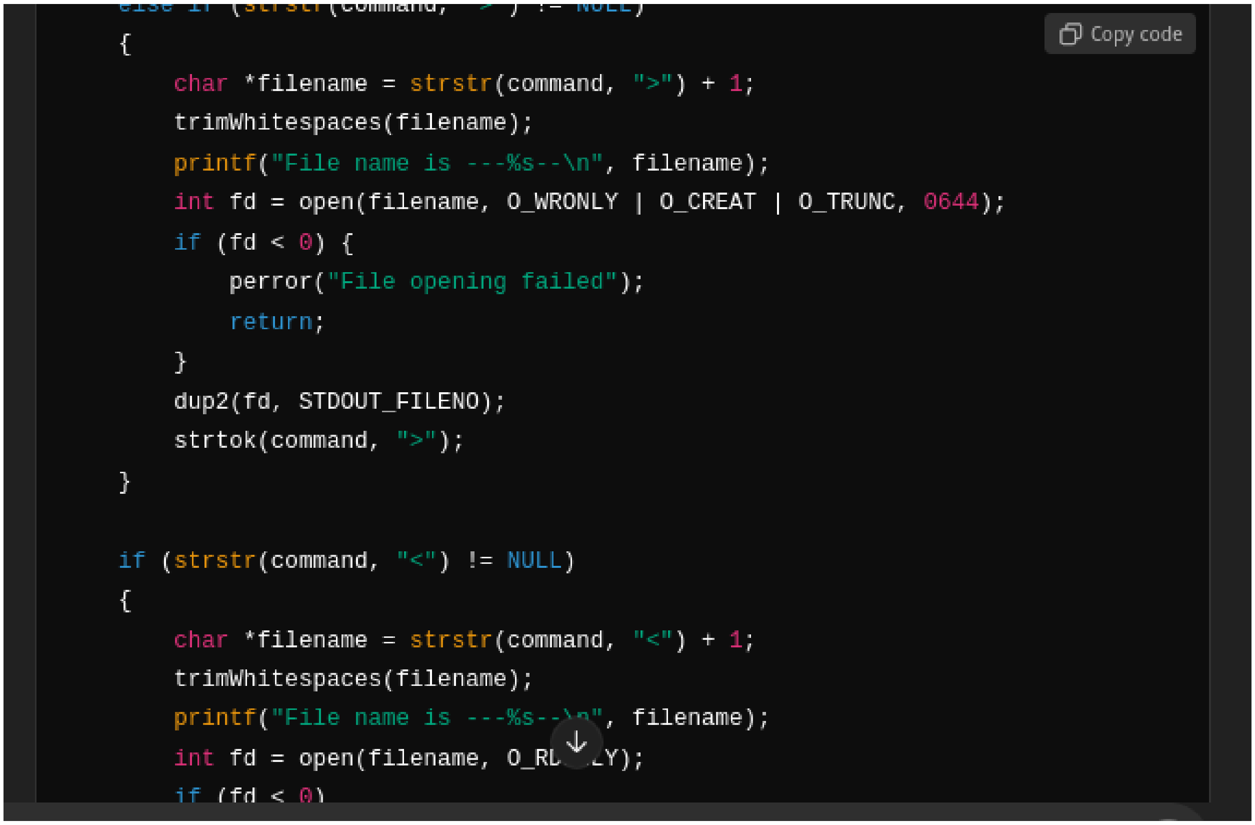
- Ensure that `dup2` is used correctly and file descriptors are correctly restored after the redirection process completes.

Here is an improved version of the `processRedirection` function that should mitigate these errors:

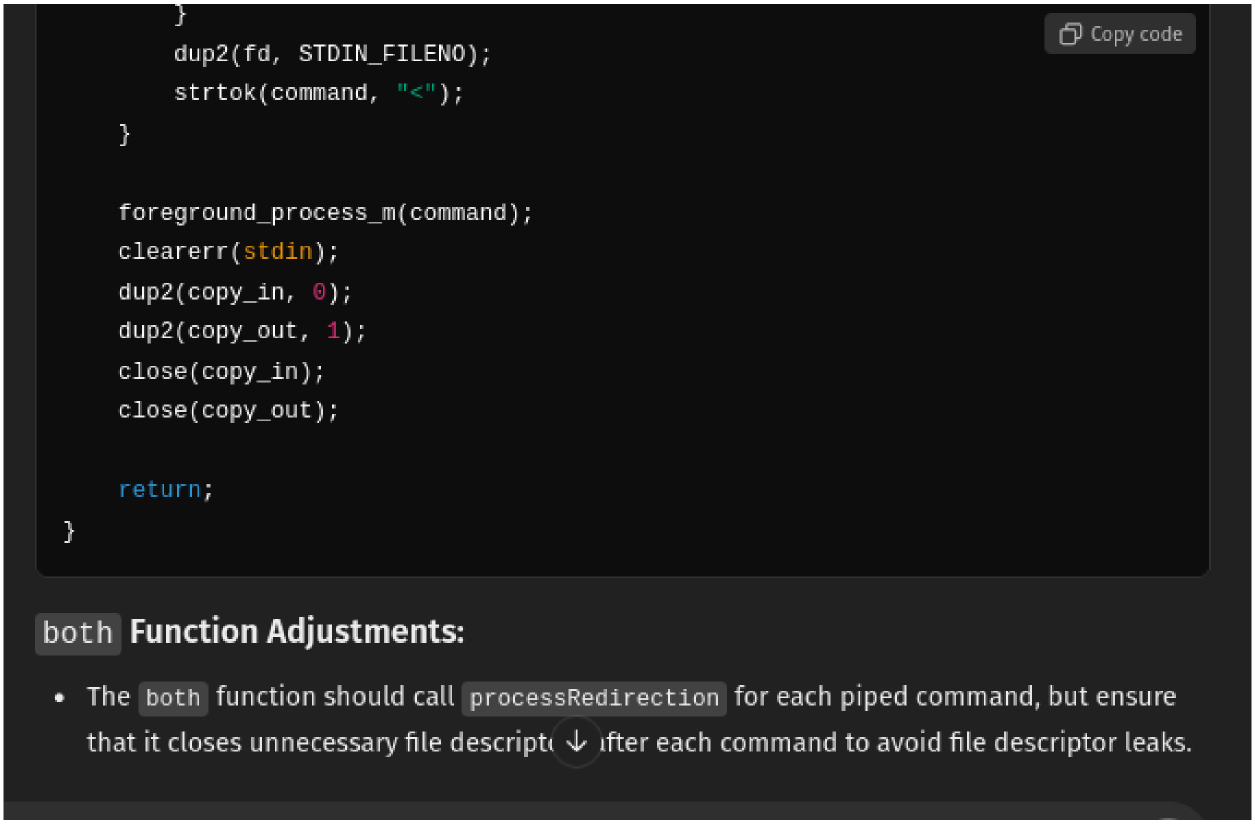
```c
void processRedirection(char *input)
{
    printf("%s\n", input);
    char *command = (char *)malloc(sizeof(char) * buf_size);
    strcpy(command, input);
    int copy_in = dup(0);
    int copy_out = dup(1);
```



```c
void processRedirection(char *input)
{
    printf("%s\n", input);
    char *command = (char *)malloc(sizeof(char) * buf_size);
    strcpy(command, input);
    int copy_in = dup(0);
    int copy_out = dup(1);

    if (strstr(command, ">>") != NULL)
    {
        char *filename = strstr(command, ">>") + 2;
        trimWhitespaces(filename);
        printf("File name is ---%s--\n", filename);
        int fd = open(filename, O_WRONLY | O_APPEND | O_CREAT, 0644);
        if (fd < 0) {
            perror("File opening failed");
            return;
        }
```



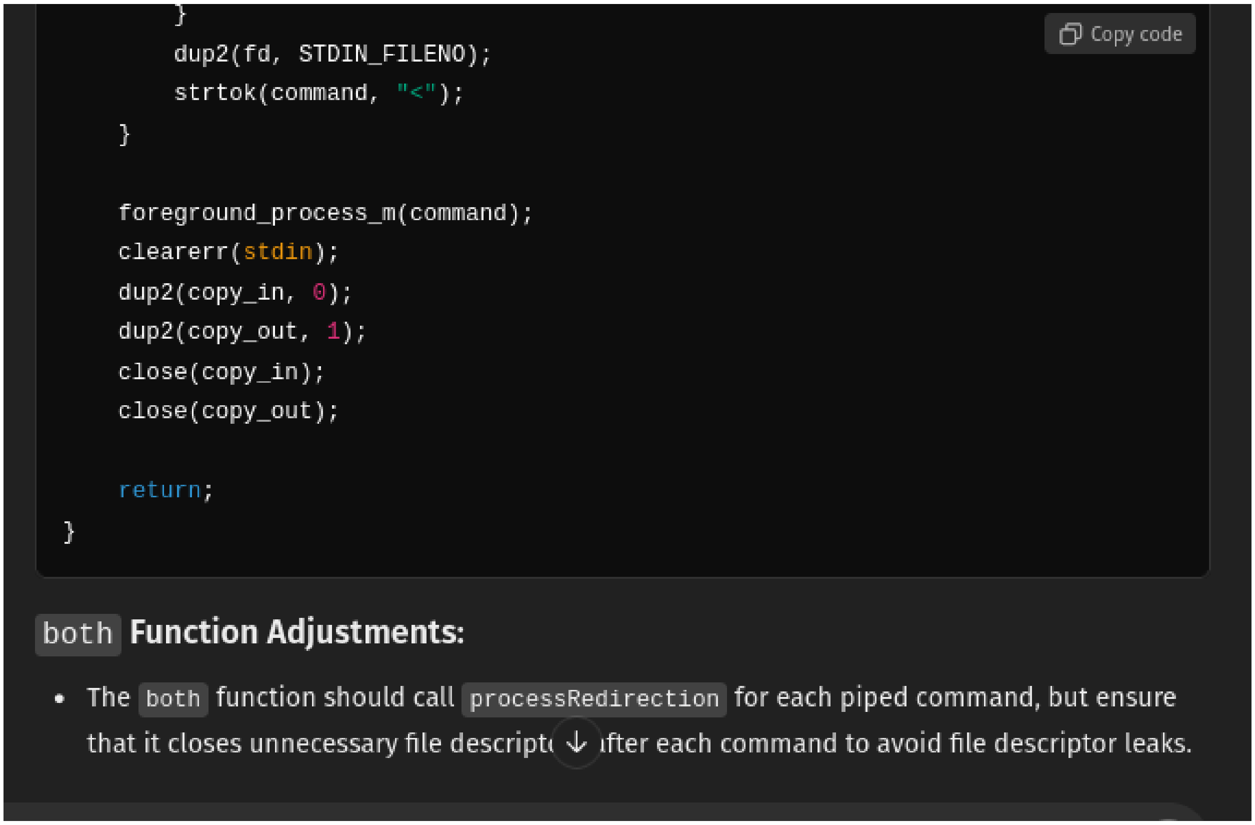
```c
    else if (strstr(command, ">") != NULL)
    {
        char *filename = strstr(command, ">") + 1;
        trimWhitespaces(filename);
        printf("File name is ---%s--\n", filename);
        int fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
        if (fd < 0) {
            perror("File opening failed");
            return;
        }
        dup2(fd, STDOUT_FILENO);
        strtok(command, ">");
    }

    if (strstr(command, "<") != NULL)
    {
        char *filename = strstr(command, "<") + 1;
        trimWhitespaces(filename);
        printf("File name is ---%s--\n", filename);
        int fd = open(filename, O_RDONLY);
        if (fd < 0)
```



```
        }
        dup2(fd, STDIN_FILENO);
        strtok(command, "<");
    }

    foreground_process_m(command);
    clearerr(stdin);
    dup2(copy_in, 0);
    dup2(copy_out, 1);
    close(copy_in);
    close(copy_out);

    return;
}
```

`both` Function Adjustments:

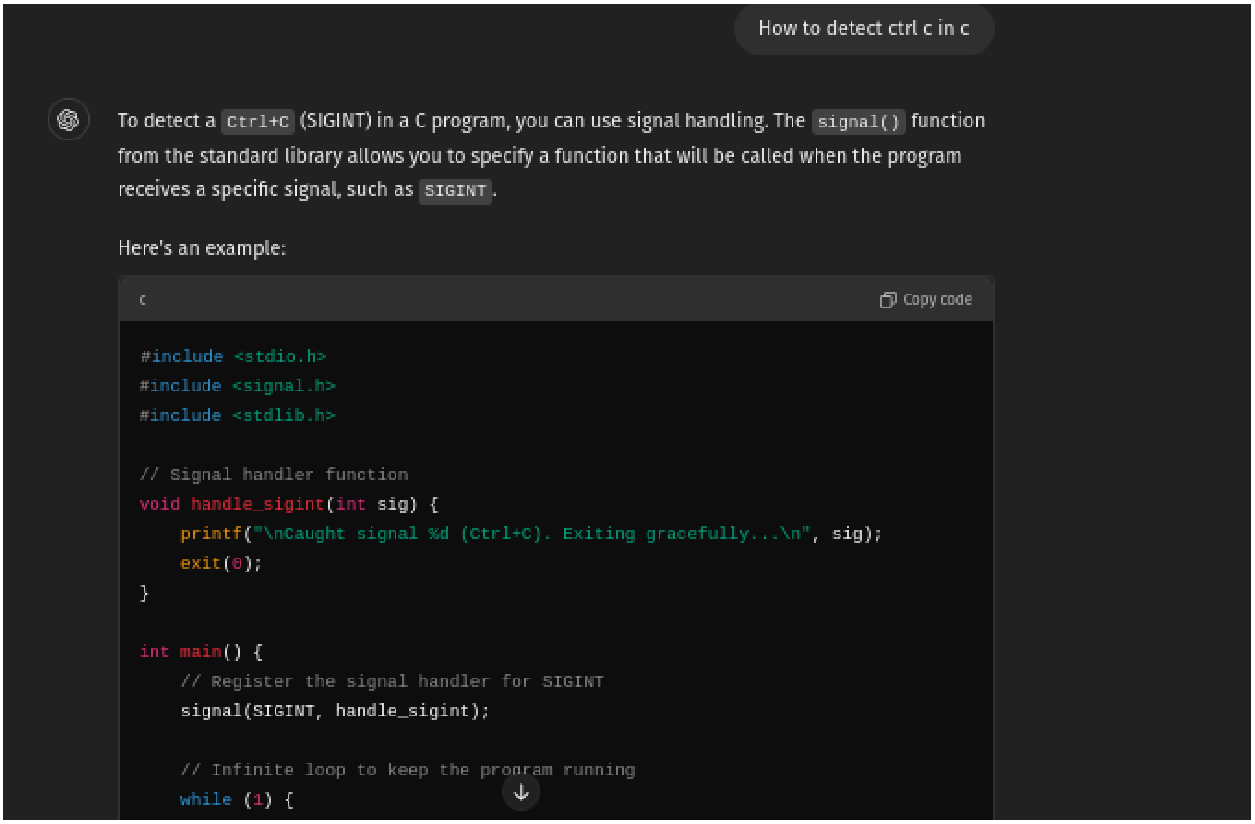
- The `both` function should call `processRedirection` for each piped command, but ensure that it closes unnecessary file descripto ↓ after each command to avoid file descriptor leaks.


Copy code

```
        }
        dup2(fd, STDIN_FILENO);
        strtok(command, "<");
    }

    foreground_process_m(command);
    clearerr(stdin);
    dup2(copy_in, 0);
    dup2(copy_out, 1);
    close(copy_in);
    close(copy_out);

    return;
}
```

`both` Function Adjustments:

- The `both` function should call `processRedirection` for each piped command, but ensure that it closes unnecessary file descripto ↓ after each command to avoid file descriptor leaks.

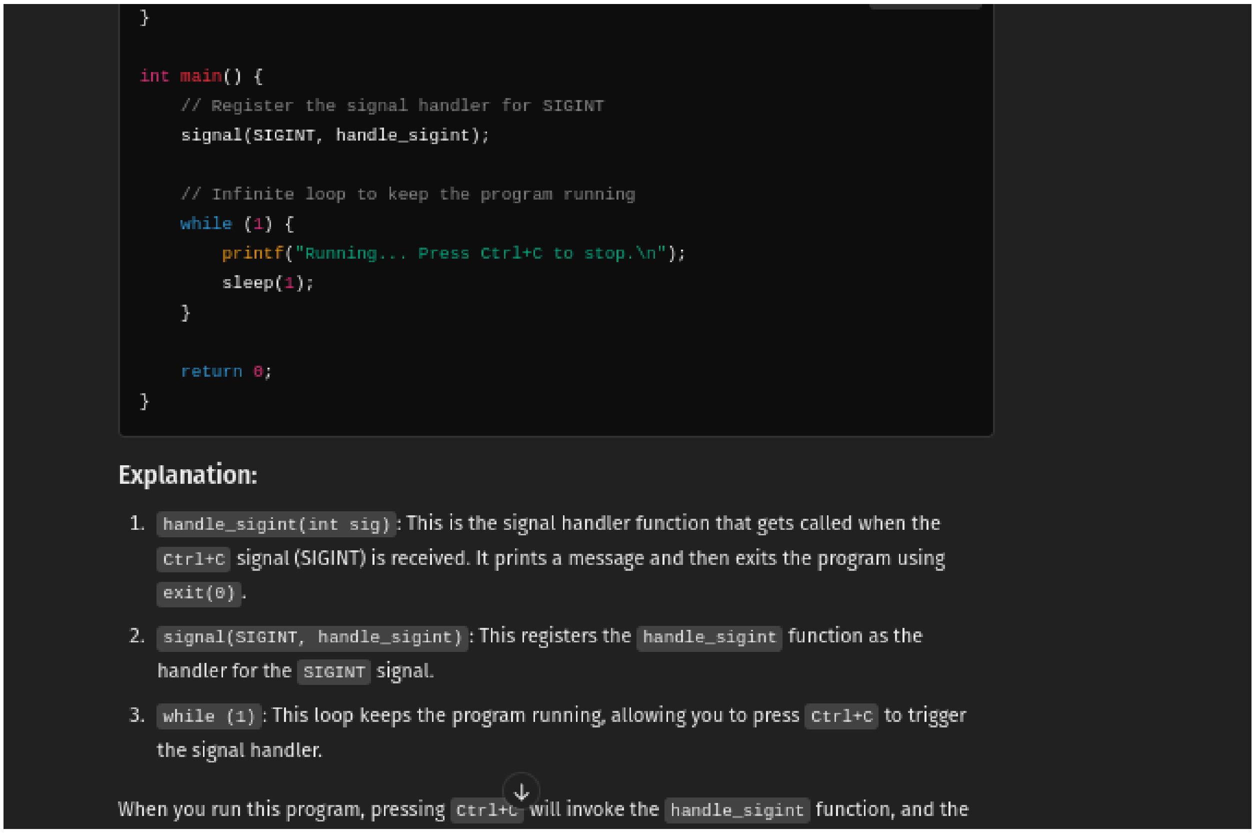


How to detect ctrl c in c

To detect a `Ctrl+C` (SIGINT) in a C program, you can use signal handling. The `signal()` function from the standard library allows you to specify a function that will be called when the program receives a specific signal, such as `SIGINT`.

Here's an example:

```c
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>

// Signal handler function
void handle_sigint(int sig) {
    printf("\nCaught signal %d (Ctrl+C). Exiting gracefully...\n", sig);
    exit(0);
}

int main() {
    // Register the signal handler for SIGINT
    signal(SIGINT, handle_sigint);

    // Infinite loop to keep the program running
    while (1) {
```



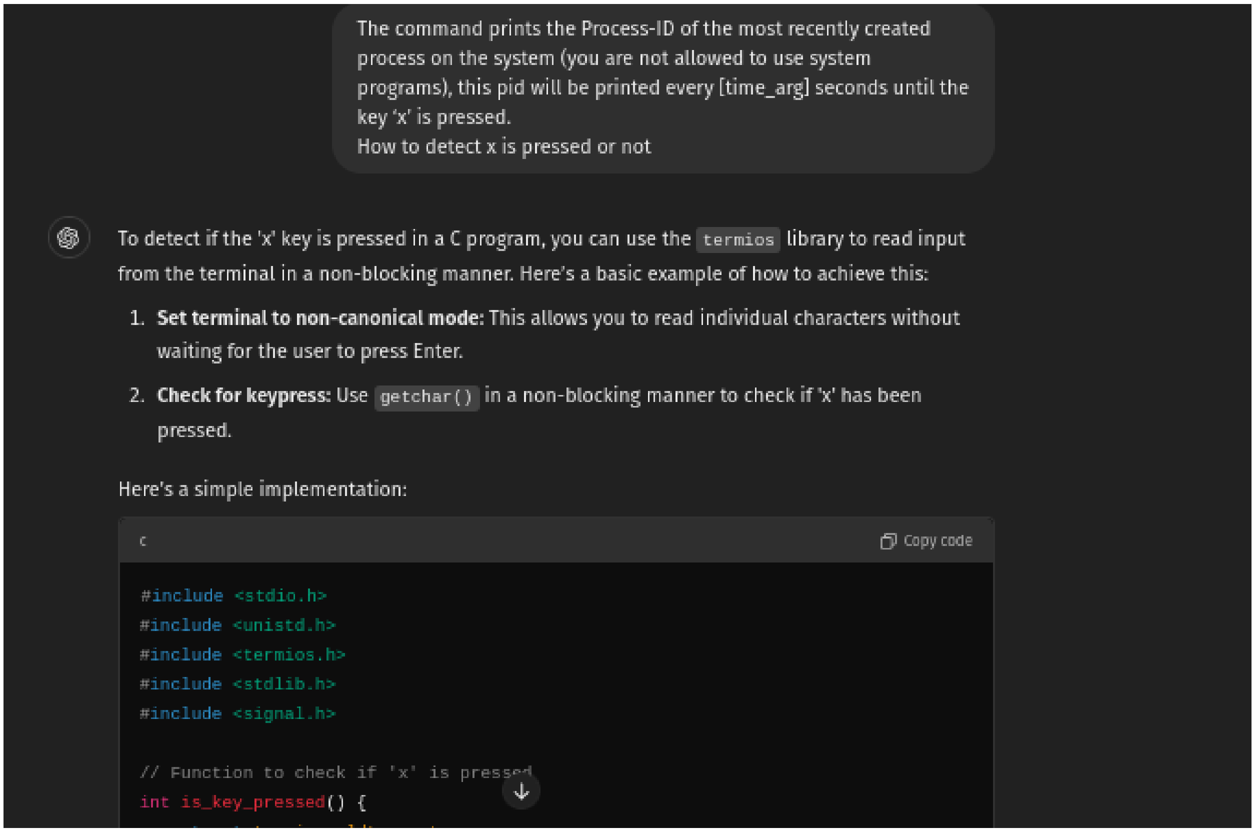
```c
}

int main() {
    // Register the signal handler for SIGINT
    signal(SIGINT, handle_sigint);

    // Infinite loop to keep the program running
    while (1) {
```

```c
        printf("Running... Press Ctrl+C to stop.\n");
        sleep(1);
    }

    return 0;
}
```

Explanation:

1. `handle_sigint(int sig)`: This is the signal handler function that gets called when the `Ctrl+C` signal (SIGINT) is received. It prints a message and then exits the program using `exit(0)`.
2. `signal(SIGINT, handle_sigint)`: This registers the `handle_sigint` function as the handler for the `SIGINT` signal.
3. `while (1)`: This loop keeps the program running, allowing you to press `Ctrl+C` to trigger the signal handler.

When you run this program, pressing `Ctrl+C` will invoke the `handle_sigint` function, and the



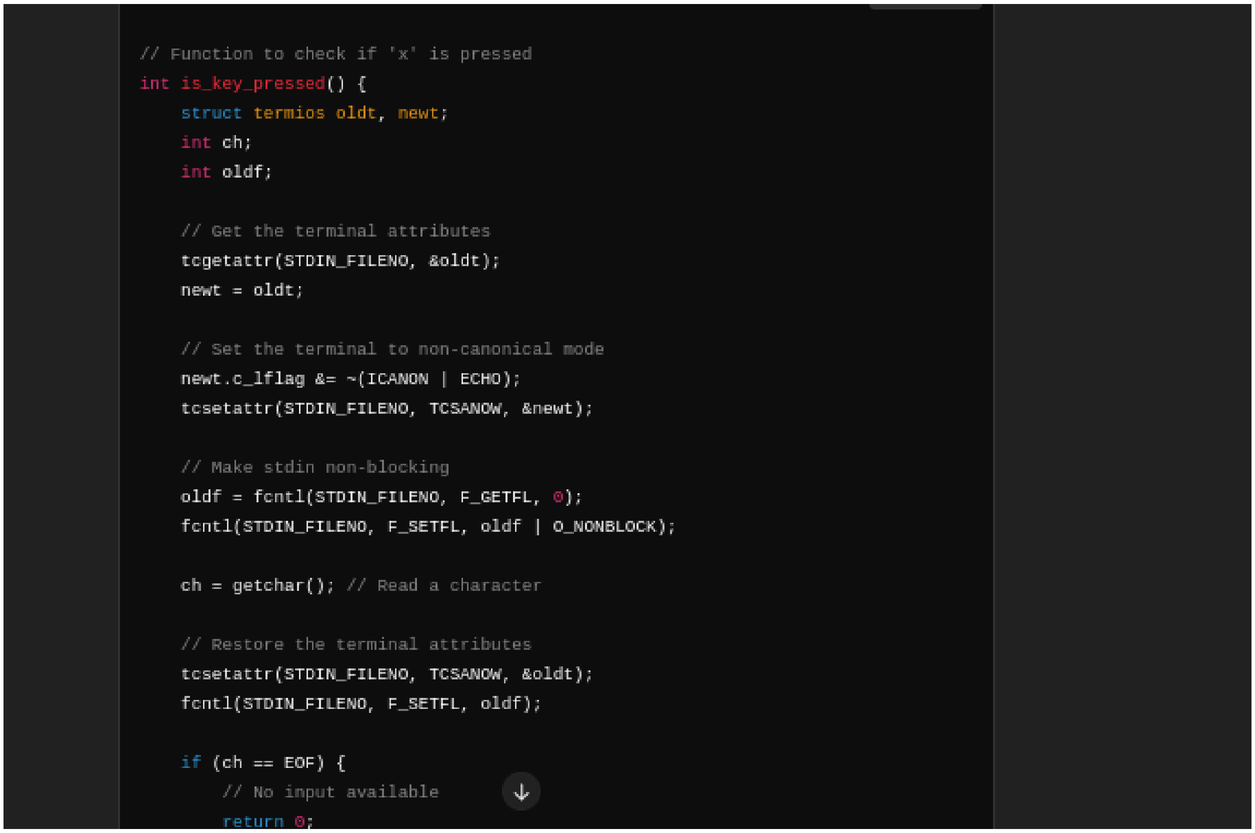
The command prints the Process-ID of the most recently created process on the system (you are not allowed to use system programs), this pid will be printed every [time_arg] seconds until the key 'x' is pressed.
How to detect x is pressed or not

To detect if the 'x' key is pressed in a C program, you can use the `termios` library to read input from the terminal in a non-blocking manner. Here's a basic example of how to achieve this:

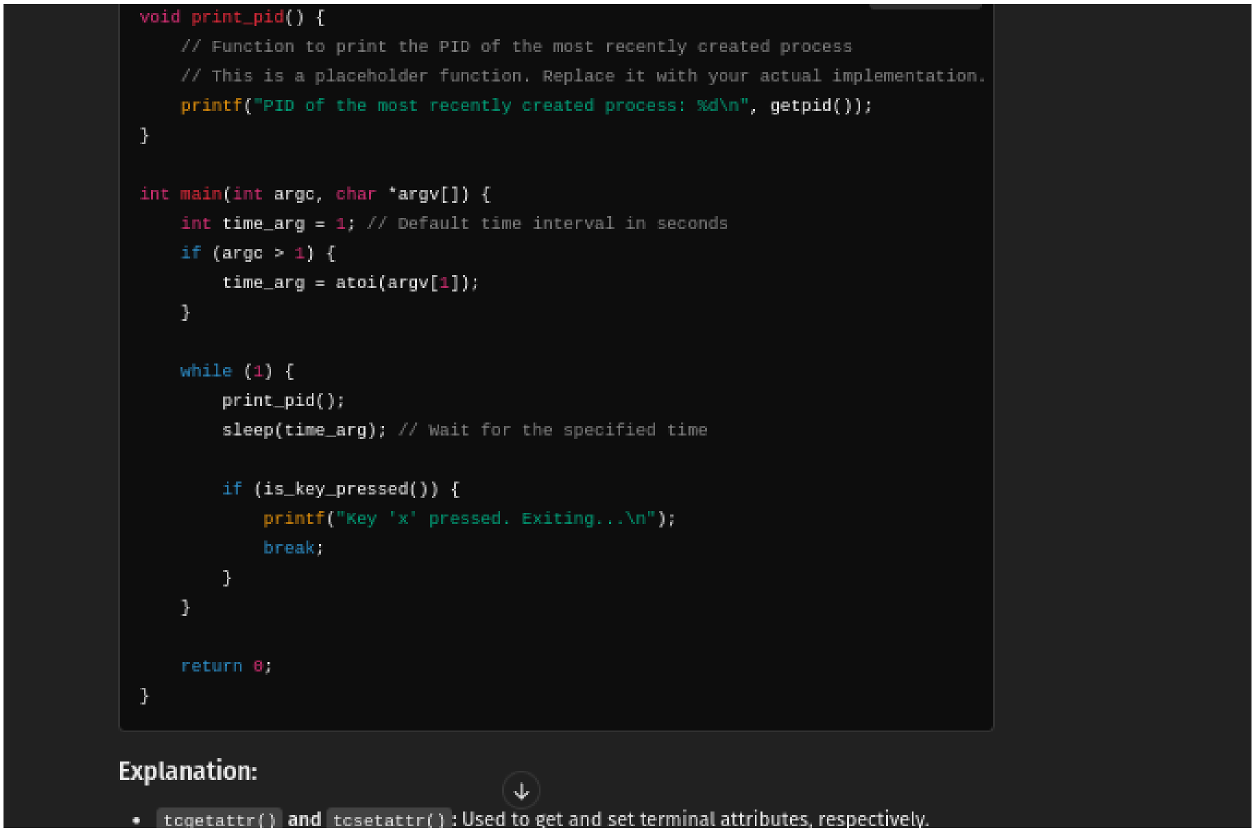
1. **Set terminal to non-canonical mode:** This allows you to read individual characters without waiting for the user to press Enter.
2. **Check for keypress:** Use `getchar()` in a non-blocking manner to check if 'x' has been pressed.

Here's a simple implementation:

```c
#include <stdio.h>
#include <unistd.h>
#include <termios.h>
#include <stdlib.h>
#include <signal.h>

// Function to check if 'x' is pressed
int is_key_pressed() {
```



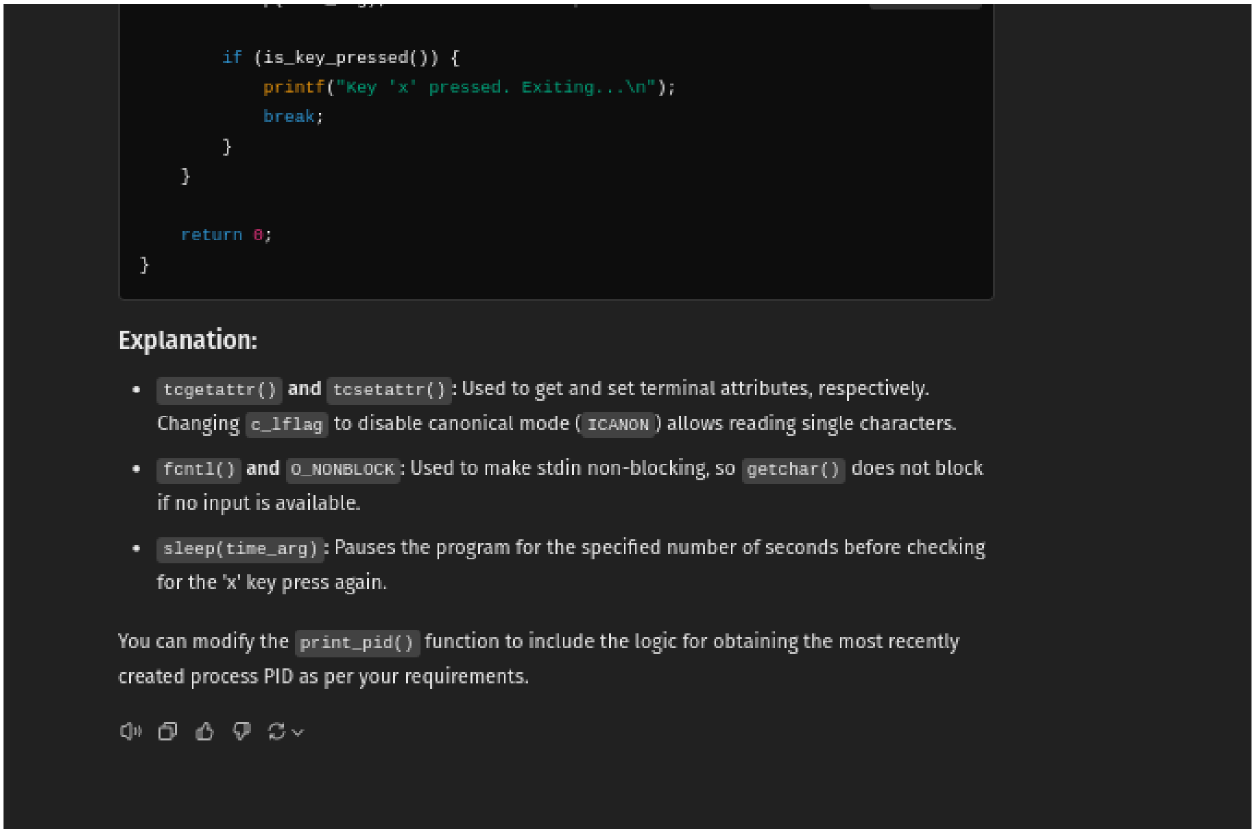
```c
// Function to check if 'x' is pressed
int is_key_pressed() {
    struct termios oldt, newt;
    int ch;
    int oldf;

    // Get the terminal attributes
    tcgetattr(STDIN_FILENO, &oldt);
    newt = oldt;

    // Set the terminal to non-canonical mode
    newt.c_lflag &= ~(ICANON | ECHO);
    tcsetattr(STDIN_FILENO, TCSANOW, &newt);

    // Make stdin non-blocking
    oldf = fcntl(STDIN_FILENO, F_GETFL, 0);
    fcntl(STDIN_FILENO, F_SETFL, oldf | O_NONBLOCK);

    ch = getchar(); // Read a character

    // Restore the terminal attributes
    tcsetattr(STDIN_FILENO, TCSANOW, &oldt);
    fcntl(STDIN_FILENO, F_SETFL, oldf);

    if (ch == EOF) {
        // No input available
        return 0;
```



```c
void print_pid() {
    // Function to print the PID of the most recently created process
    // This is a placeholder function. Replace it with your actual implementation.
    printf("PID of the most recently created process: %d\n", getpid());
}

int main(int argc, char *argv[]) {
    int time_arg = 1; // Default time interval in seconds
    if (argc > 1) {
        time_arg = atoi(argv[1]);
    }

    while (1) {
        print_pid();
        sleep(time_arg); // Wait for the specified time

        if (is_key_pressed()) {
            printf("Key 'x' pressed. Exiting...\n");
            break;
        }
    }

    return 0;
}
```

Explanation:

- `tcgetattr()` **and** `tcsetattr()`: Used to get and set terminal attributes, respectively.



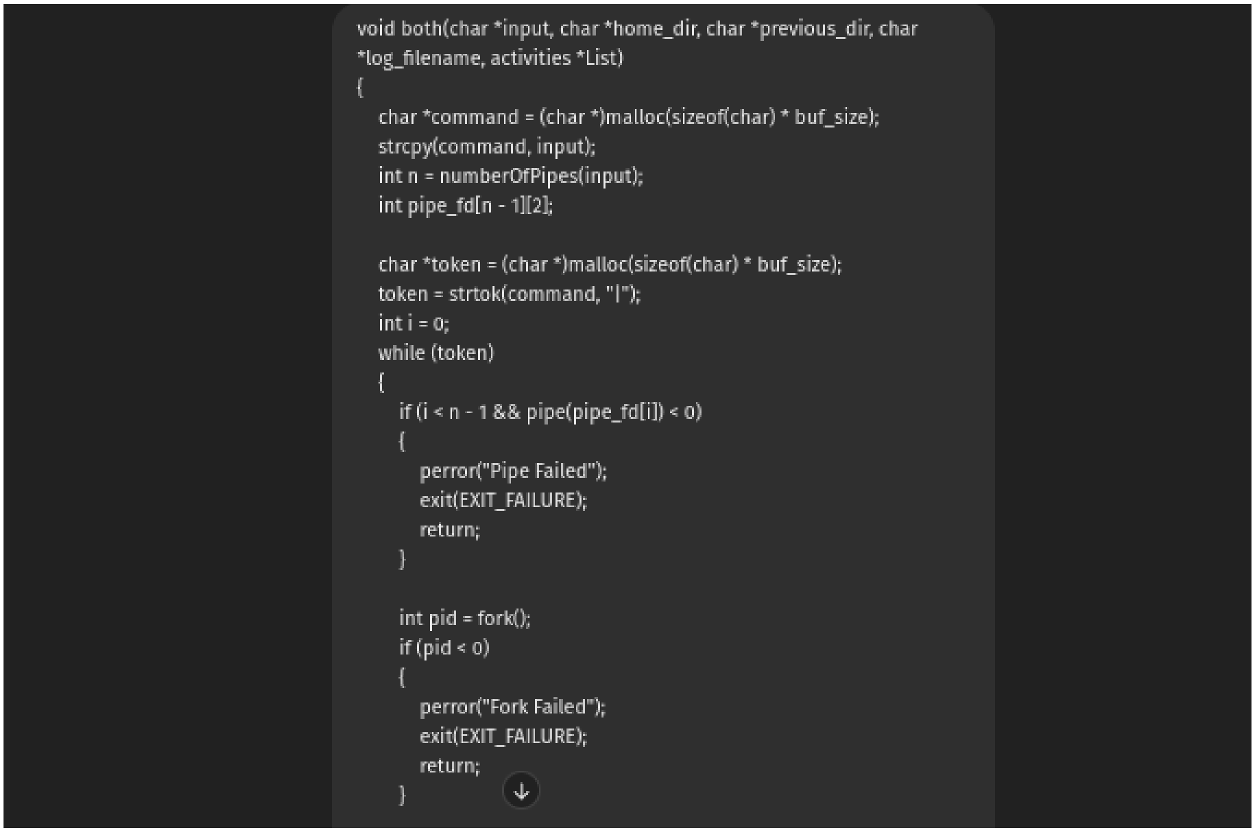
```c
        if (is_key_pressed()) {
            printf("Key 'x' pressed. Exiting...\n");
            break;
        }
    }

    return 0;
}
```

Explanation:

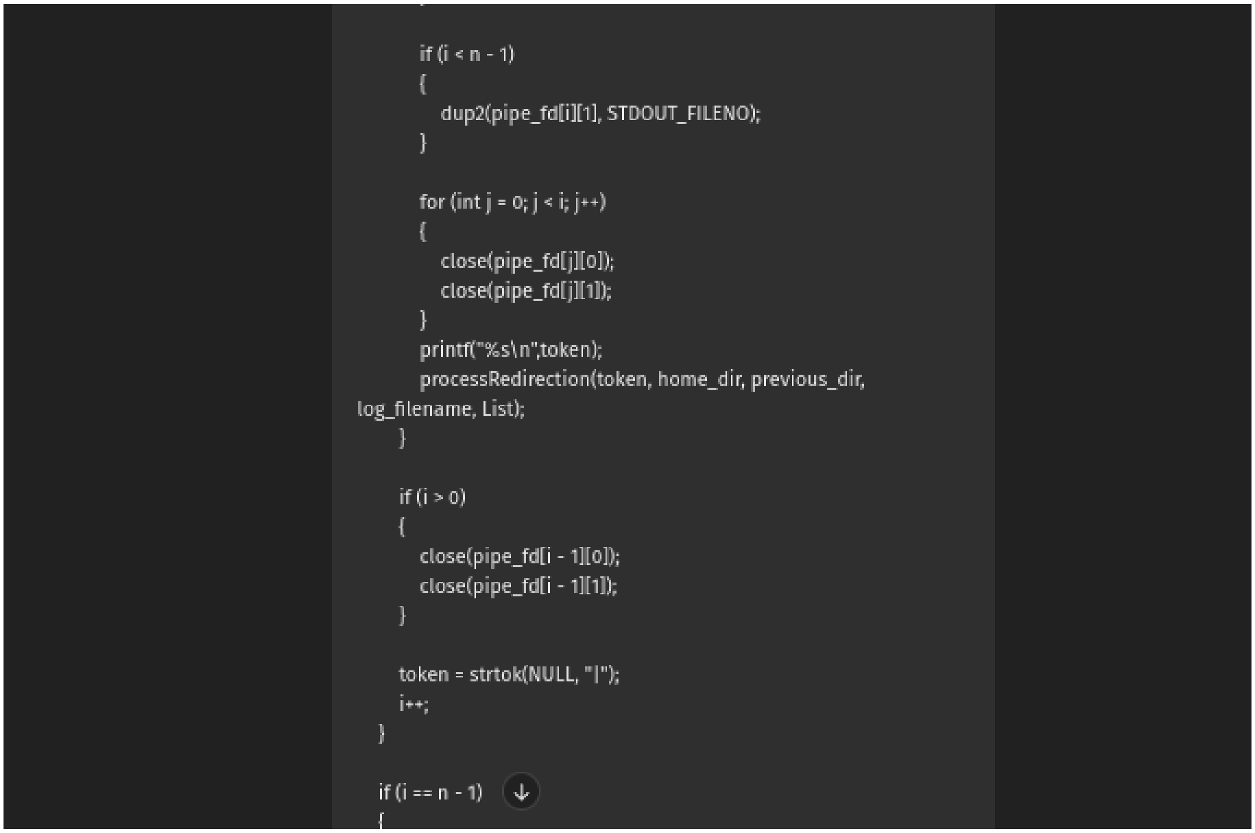
- `tcgetattr()` **and** `tcsetattr()`: Used to get and set terminal attributes, respectively.

Changing `c_lflag` to disable canonical mode (`ICANON`) allows reading single characters.
- `fcntl()` **and** `O_NONBLOCK`: Used to make stdin non-blocking, so `getchar()` does not block if no input is available.
- `sleep(time_arg)`: Pauses the program for the specified number of seconds before checking for the 'x' key press again.

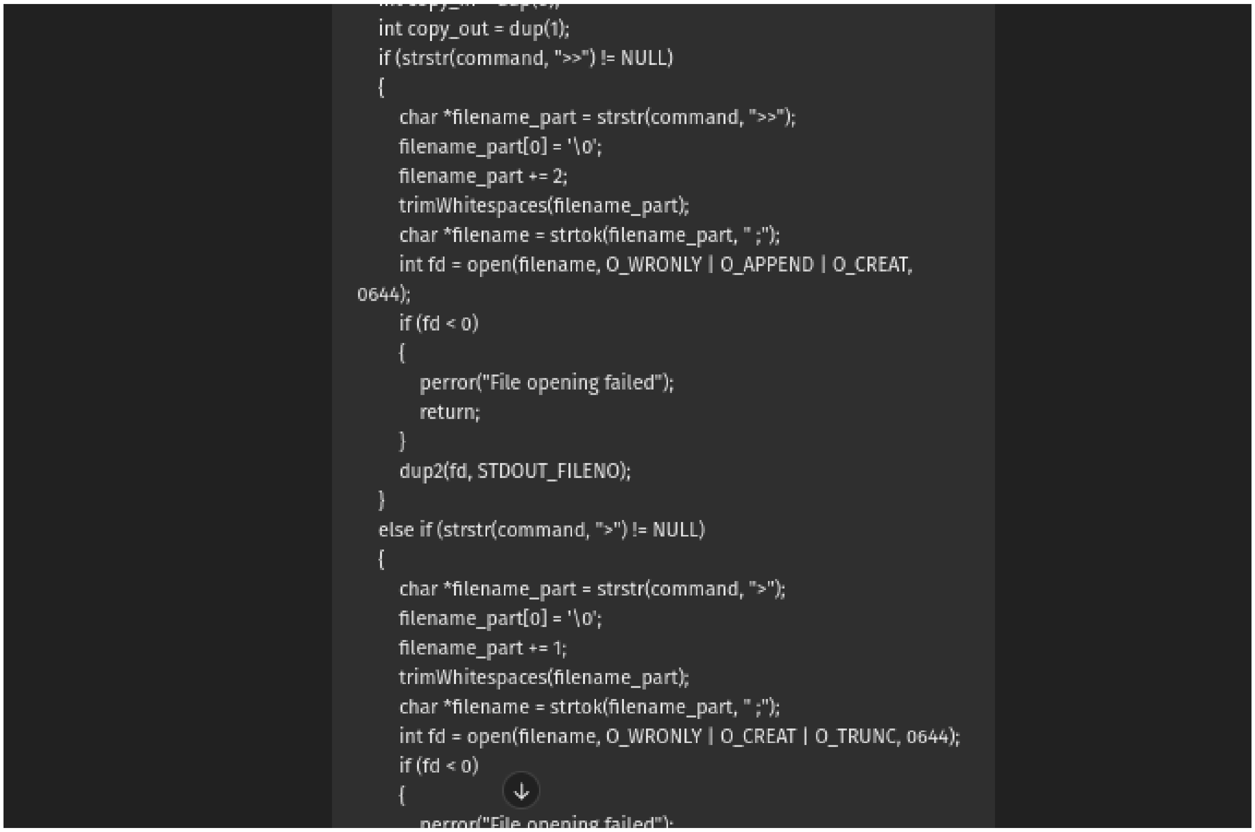
You can modify the `print_pid()` function to include the logic for obtaining the most recently created process PID as per your requirements.



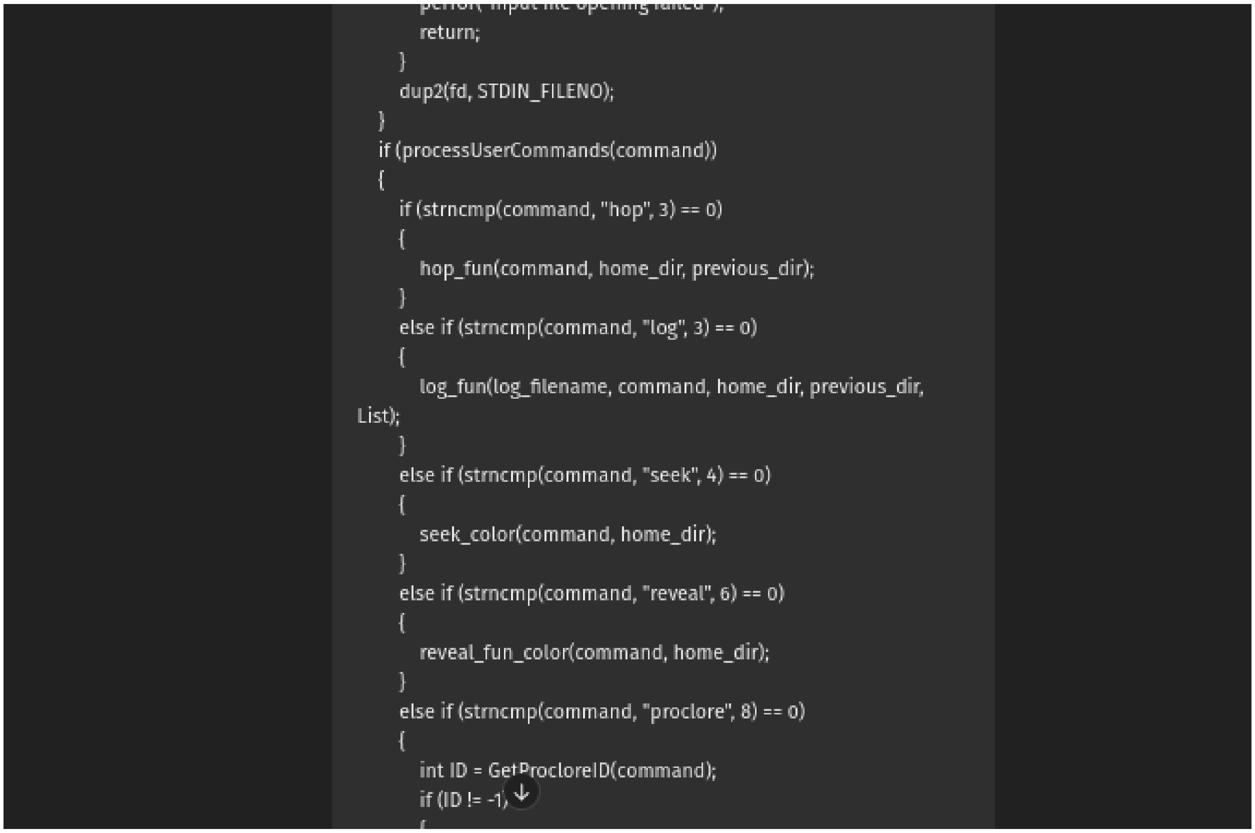
```
void both(char *input, char *home_dir, char *previous_dir, char
*log_filename, activities *List)
{
    char *command = (char *)malloc(sizeof(char) * buf_size);
    strcpy(command, input);
    int n = numberOfPipes(input);
    int pipe_fd[n - 1][2];

    char *token = (char *)malloc(sizeof(char) * buf_size);
    token = strtok(command, "|");
    int i = 0;
    while (token)
    {
        if (i < n - 1 && pipe(pipe_fd[i]) < 0)
        {
            perror("Pipe Failed");
            exit(EXIT_FAILURE);
            return;
        }

        int pid = fork();
        if (pid < 0)
        {
            perror("Fork Failed");
            exit(EXIT_FAILURE);
            return;
        }
```



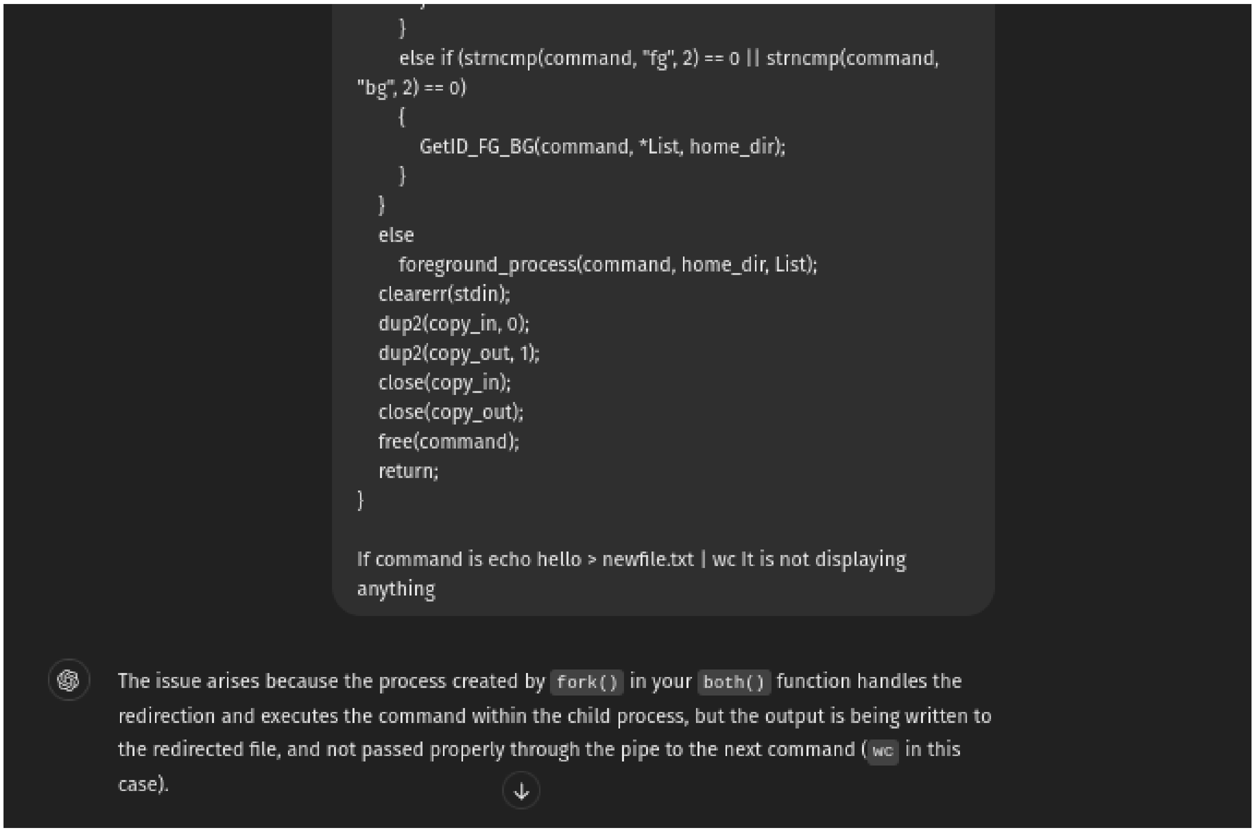
```
            if (i < n - 1)
            {
                dup2(pipe_fd[i][1], STDOUT_FILENO);
            }

            for (int j = 0; j < i; j++)
            {
                close(pipe_fd[j][0]);
                close(pipe_fd[j][1]);
            }
            printf("%s\n",token);
            processRedirection(token, home_dir, previous_dir,
log_filename, List);
        }

        if (i > 0)
        {
            close(pipe_fd[i - 1][0]);
            close(pipe_fd[i - 1][1]);
        }

        token = strtok(NULL, "|");
        i++;
    }

    if (i == n - 1)
    {
```



```
    int copy_out = dup(1);
    if (strstr(command, ">>") != NULL)
    {
        char *filename_part = strstr(command, ">>");
        filename_part[0] = '\0';
        filename_part += 2;
        trimWhitespaces(filename_part);
        char *filename = strtok(filename_part, " ;");
        int fd = open(filename, O_WRONLY | O_APPEND | O_CREAT,
0644);
        if (fd < 0)
        {
            perror("File opening failed");
            return;
        }
        dup2(fd, STDOUT_FILENO);
    }
    else if (strstr(command, ">") != NULL)
    {
        char *filename_part = strstr(command, ">");
        filename_part[0] = '\0';
        filename_part += 1;
        trimWhitespaces(filename_part);
        char *filename = strtok(filename_part, " ;");
        int fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
        if (fd < 0)
        {
            perror("File opening failed");
```



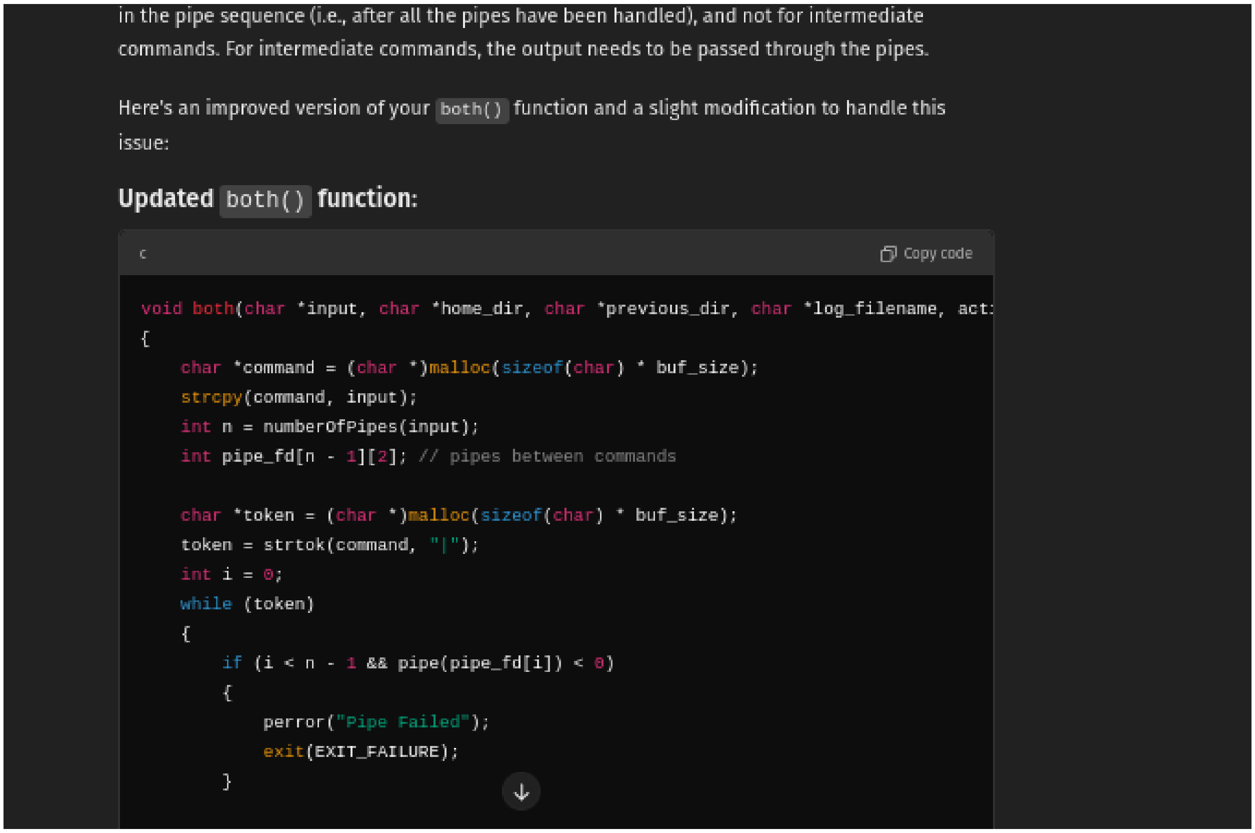
```
            perror("Input file opening failed");
            return;
        }
        dup2(fd, STDIN_FILENO);
    }
    if (processUserCommands(command))
    {
        if (strncmp(command, "hop", 3) == 0)
        {
            hop_fun(command, home_dir, previous_dir);
        }
        else if (strncmp(command, "log", 3) == 0)
        {
            log_fun(log_filename, command, home_dir, previous_dir,
List);
        }
        else if (strncmp(command, "seek", 4) == 0)
        {
            seek_color(command, home_dir);
        }
        else if (strncmp(command, "reveal", 6) == 0)
        {
            reveal_fun_color(command, home_dir);
        }
        else if (strncmp(command, "proclore", 8) == 0)
        {
            int ID = GetProcloreID(command);
            if (ID != -1)
            {
```



```
        }
        else if (strncmp(command, "fg", 2) == 0 || strncmp(command,
"bg", 2) == 0)
        {
            GetID_FG_BG(command, *List, home_dir);
        }
    }
    else
        foreground_process(command, home_dir, List);
    clearerr(stdin);
    dup2(copy_in, 0);
    dup2(copy_out, 1);
    close(copy_in);
    close(copy_out);
    free(command);
    return;
}
```

If command is echo hello > newfile.txt | wc It is not displaying anything

The issue arises because the process created by `fork()` in your `both()` function handles the redirection and executes the command within the child process, but the output is being written to the redirected file, and not passed properly through the pipe to the next command (`wc` in this case).

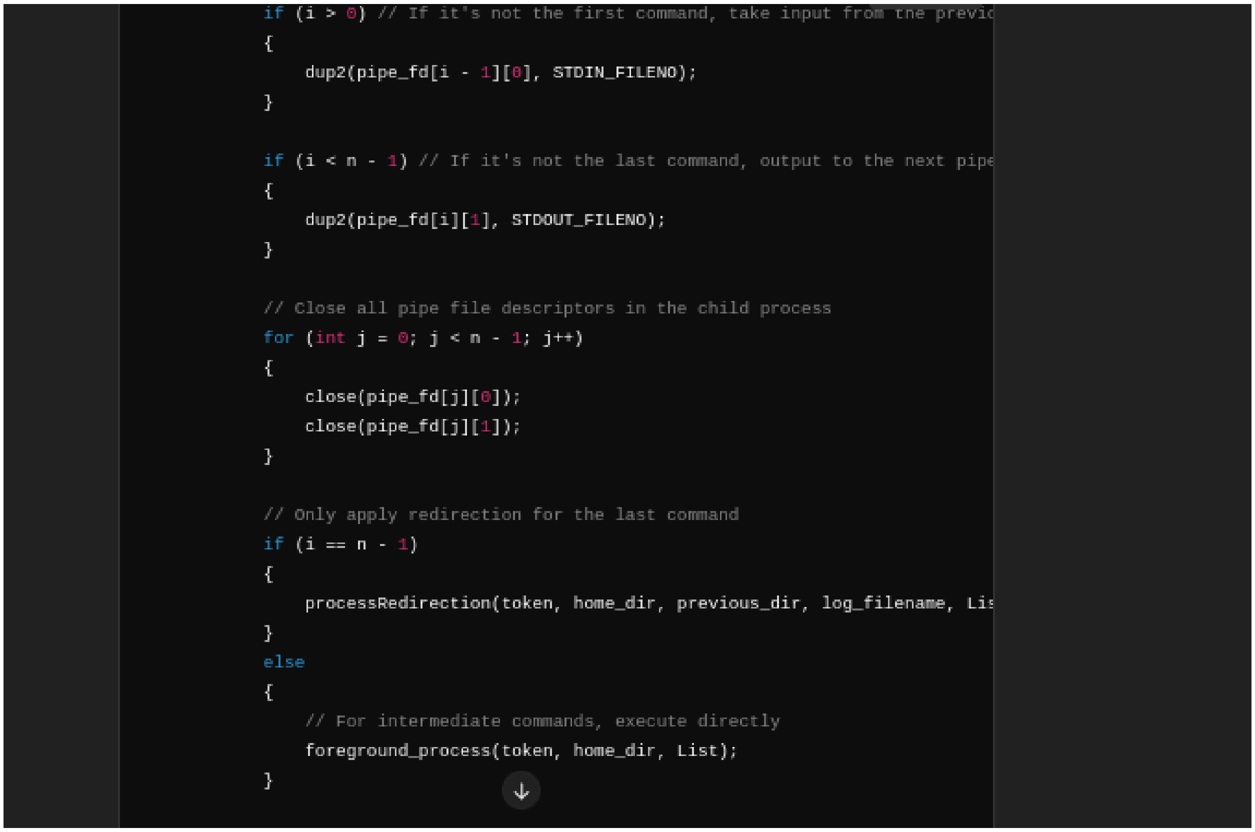


in the pipe sequence (i.e., after all the pipes have been handled), and not for intermediate commands. For intermediate commands, the output needs to be passed through the pipes.

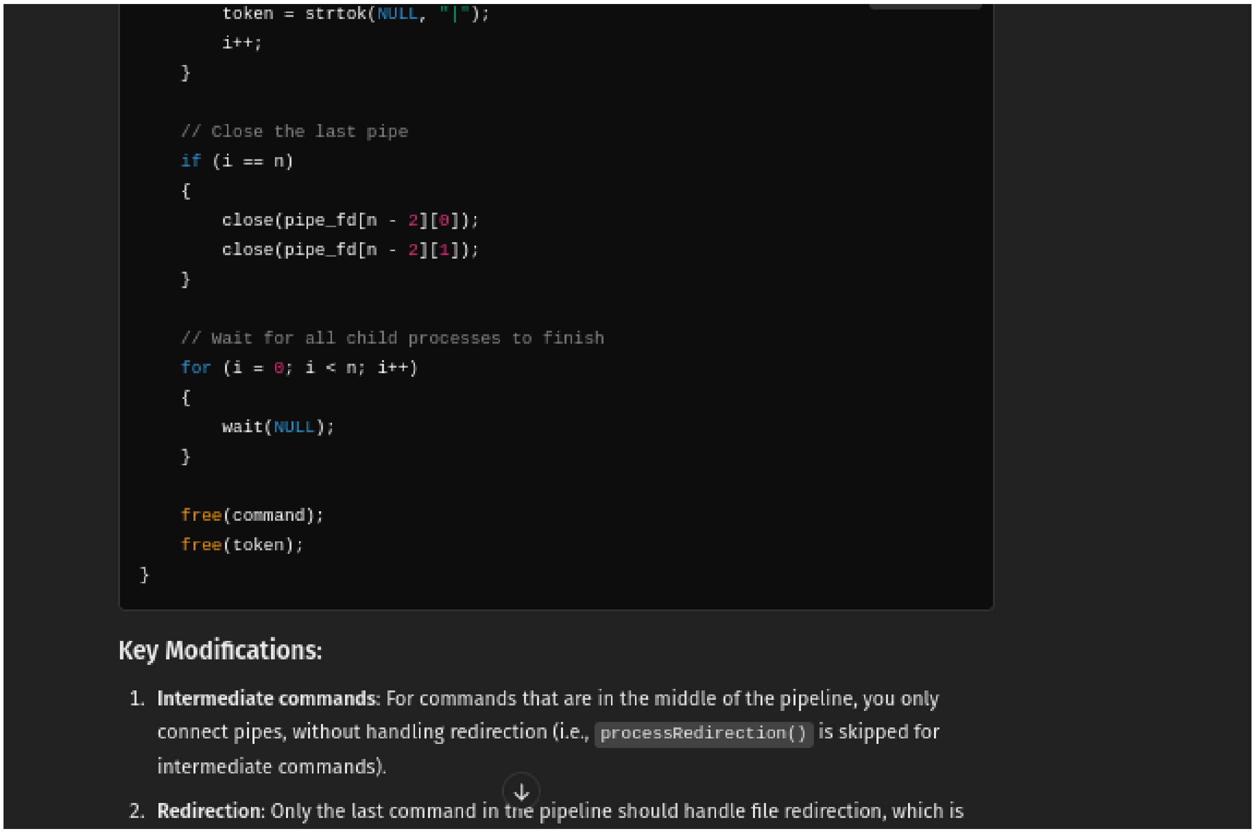
Here's an improved version of your `both()` function and a slight modification to handle this issue:

Updated `both()` function:

```c
void both(char *input, char *home_dir, char *previous_dir, char *log_filename, act
{
    char *command = (char *)malloc(sizeof(char) * buf_size);
    strcpy(command, input);
    int n = numberOfPipes(input);
    int pipe_fd[n - 1][2]; // pipes between commands

    char *token = (char *)malloc(sizeof(char) * buf_size);
    token = strtok(command, "|");
    int i = 0;
    while (token)
    {
        if (i < n - 1 && pipe(pipe_fd[i]) < 0)
        {
            perror("Pipe Failed");
            exit(EXIT_FAILURE);
        }
```



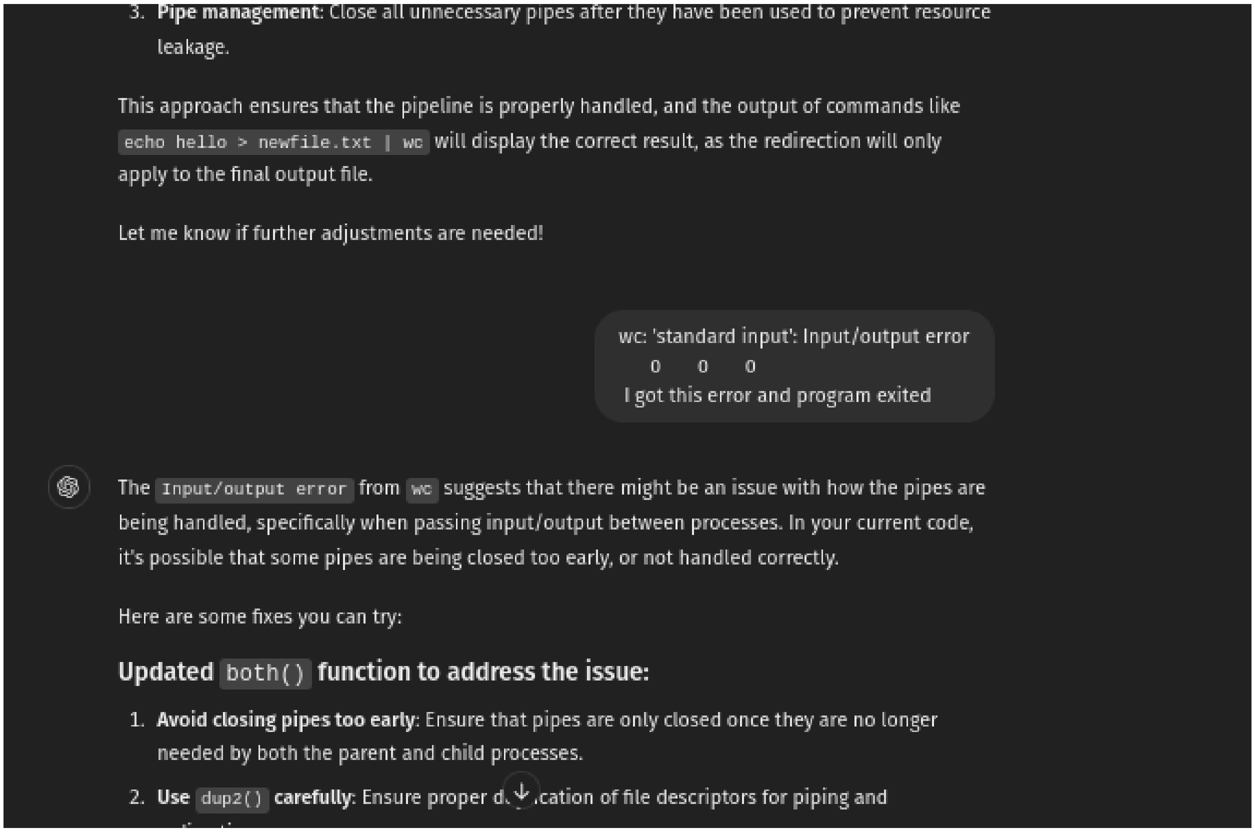
```
            if (i > 0) // If it's not the first command, take input from the previo
            {
                dup2(pipe_fd[i - 1][0], STDIN_FILENO);
            }

            if (i < n - 1) // If it's not the last command, output to the next pipe
            {
                dup2(pipe_fd[i][1], STDOUT_FILENO);
            }

            // Close all pipe file descriptors in the child process
            for (int j = 0; j < n - 1; j++)
            {
                close(pipe_fd[j][0]);
                close(pipe_fd[j][1]);
            }

            // Only apply redirection for the last command
            if (i == n - 1)
            {
                processRedirection(token, home_dir, previous_dir, log_filename, Lis
            }
            else
            {
                // For intermediate commands, execute directly
                foreground_process(token, home_dir, List);
            }
```



```
        token = strtok(NULL, "|");
        i++;
    }

    // Close the last pipe
    if (i == n)
    {
        close(pipe_fd[n - 2][0]);
        close(pipe_fd[n - 2][1]);
    }

    // Wait for all child processes to finish
    for (i = 0; i < n; i++)
    {
        wait(NULL);
    }

    free(command);
    free(token);
}
```

Key Modifications:

1. **Intermediate commands**: For commands that are in the middle of the pipeline, you only connect pipes, without handling redirection (i.e., `processRedirection()` is skipped for intermediate commands).
2. **Redirection**: Only the last command in the pipeline should handle file redirection, which is



3. **Pipe management**: Close all unnecessary pipes after they have been used to prevent resource leakage.

This approach ensures that the pipeline is properly handled, and the output of commands like `echo hello > newfile.txt | wc` will display the correct result, as the redirection will only apply to the final output file.

Let me know if further adjustments are needed!

```
wc: 'standard input': Input/output error
      0       0       0
```

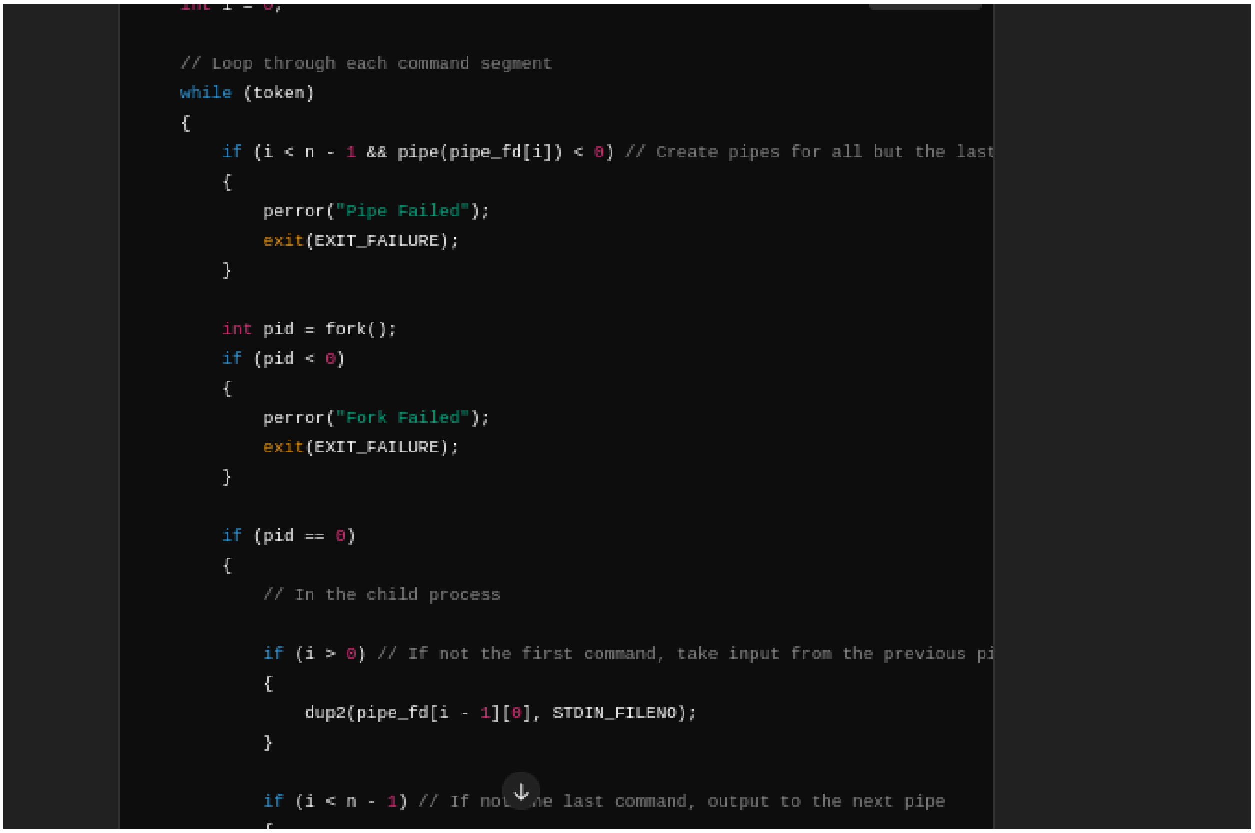
I got this error and program exited

The `Input/output error` from `wc` suggests that there might be an issue with how the pipes are being handled, specifically when passing input/output between processes. In your current code, it's possible that some pipes are being closed too early, or not handled correctly.

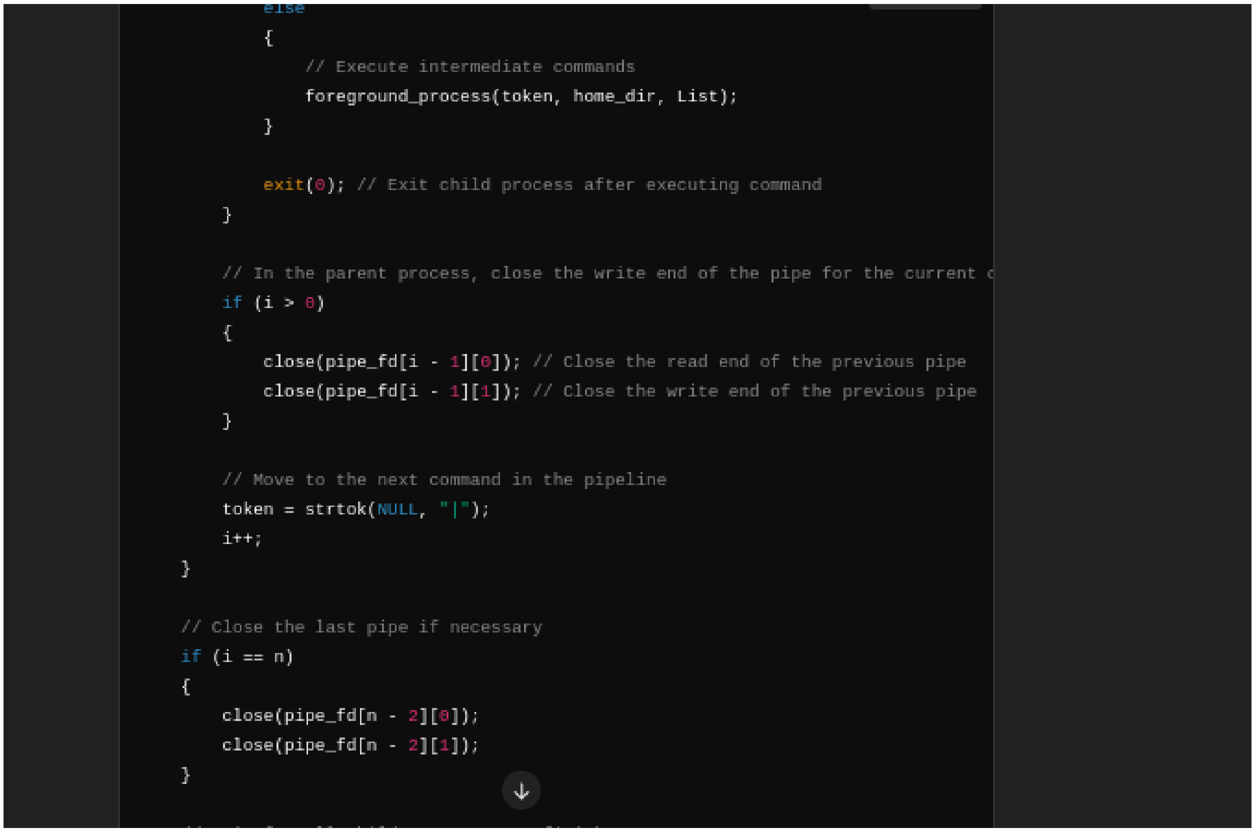
Here are some fixes you can try:

Updated `both()` function to address the issue:

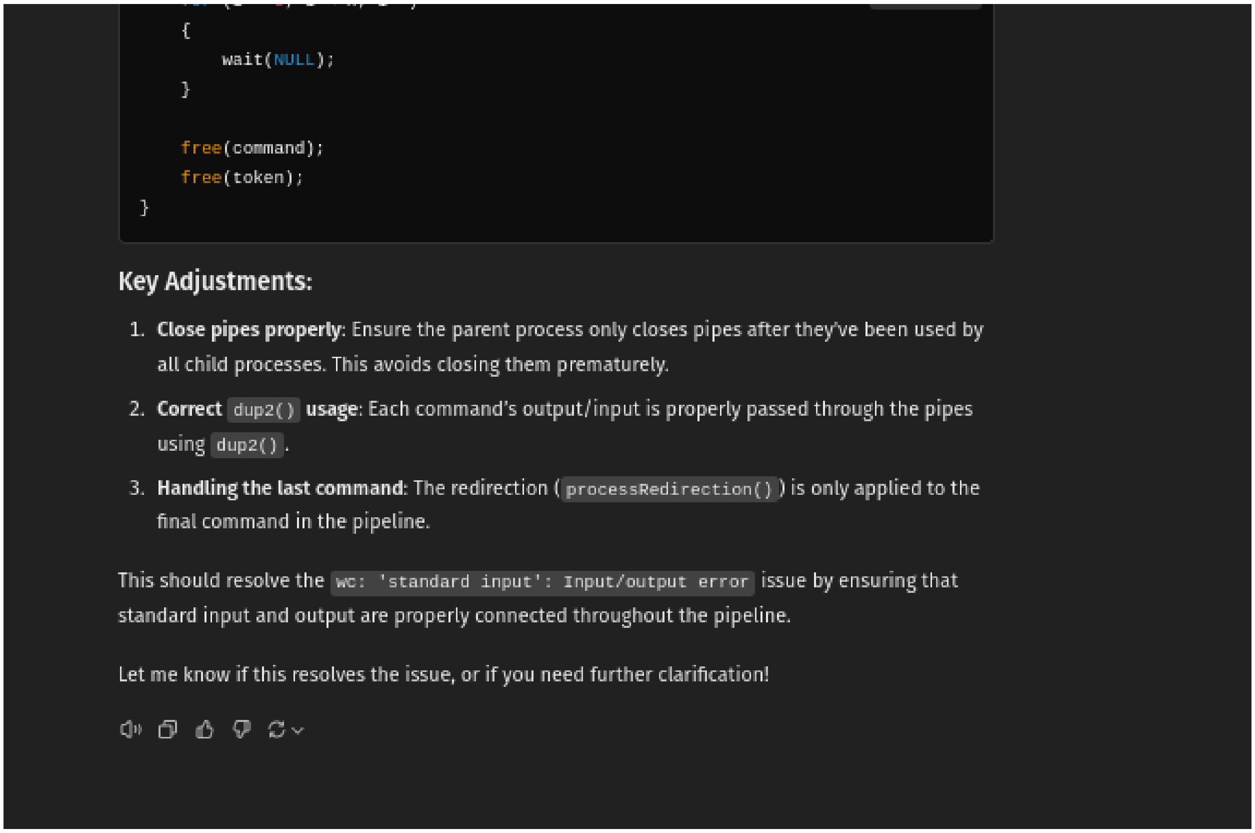
1. **Avoid closing pipes too early**: Ensure that pipes are only closed once they are no longer needed by both the parent and child processes.
2. **Use `dup2()` carefully**: Ensure proper d cation of file descriptors for piping and



```
    int i = 0;

    // Loop through each command segment
    while (token)
    {
        if (i < n - 1 && pipe(pipe_fd[i]) < 0) // Create pipes for all but the last
        {
            perror("Pipe Failed");
            exit(EXIT_FAILURE);
        }

        int pid = fork();
        if (pid < 0)
        {
            perror("Fork Failed");
            exit(EXIT_FAILURE);
        }

        if (pid == 0)
        {
            // In the child process

            if (i > 0) // If not the first command, take input from the previous pi
            {
                dup2(pipe_fd[i - 1][0], STDIN_FILENO);
            }

            if (i < n - 1) // If not the last command, output to the next pipe
            {
```



```
            else
            {
                // Execute intermediate commands
                foreground_process(token, home_dir, List);
            }

            exit(0); // Exit child process after executing command
        }

        // In the parent process, close the write end of the pipe for the current c
        if (i > 0)
        {
            close(pipe_fd[i - 1][0]); // Close the read end of the previous pipe
            close(pipe_fd[i - 1][1]); // Close the write end of the previous pipe
        }

        // Move to the next command in the pipeline
        token = strtok(NULL, "|");
        i++;
    }

    // Close the last pipe if necessary
    if (i == n)
    {
        close(pipe_fd[n - 2][0]);
        close(pipe_fd[n - 2][1]);
    }
```



```
    {
        wait(NULL);
    }

    free(command);
    free(token);
}
```

Key Adjustments:

1. **Close pipes properly**: Ensure the parent process only closes pipes after they've been used by all child processes. This avoids closing them prematurely.
2. **Correct `dup2()` usage**: Each command's output/input is properly passed through the pipes using `dup2()`.
3. **Handling the last command**: The redirection (`processRedirection()`) is only applied to the final command in the pipeline.

This should resolve the `wc: 'standard input': Input/output error` issue by ensuring that standard input and output are properly connected throughout the pipeline.

Let me know if this resolves the issue, or if you need further clarification!