Документация к файлу executor.c
Исполнитель команд

Разработчик: [Ваше имя]

16 ноября 2025 г.

Аннотация

Данный документ описывает структуру и функциональность модуля выполнения команд, реализованного в файле `executor.c`. Модуль отвечает за выполнение внешних и встроенных команд, управление процессами, обработку конвейеров, последовательностей команд и перенаправлений ввода/вывода.

Содержание

1 Обзор модуля **2**

2 Структура модуля **2**
2.1 Заголовочные файлы 2

3 Функции управления перенаправлениями **2**
3.1 Функция `apply_redirections` 2

4 Функции поиска и выполнения команд **4**
4.1 Функция `get_full_path` 4
4.2 Функция `execute_external` 5

5 Функции работы с конвейерами **7**
5.1 Функция `split_pipeline` 7
5.2 Функция `execute_pipeline` 7

6 Функции работы с последовательностями команд **10**
6.1 Функция `execute_command_sequence` 10

7 Главная функция выполнения **11**
7.1 Функция `execute_command` 11

8 Примеры выполнения **13**
8.1 Пример 1: Простая внешняя команда 13
8.2 Пример 2: Конвейер 13
8.3 Пример 3: Последовательность с логическими операторами 13
8.4 Пример 4: Команда с перенаправлениями 14

9 Особенности реализации **14**
9.1 Управление процессами 14
9.2 Управление файловыми дескрипторами 14
9.3 Обработка ошибок 14
9.4 Управление памятью 14

10 Взаимодействие с другими модулями **14**
10.1 Зависимости 14
10.2 Поток данных 15

11 Заключение **15**

1 Обзор модуля

Файл `executor.c` реализует сложную систему выполнения команд, включающую:

- Выполнение внешних команд через поиск в PATH
- Обработку встроенных команд shell'а
- Управление перенаправлениями ввода/вывода
- Реализацию конвейеров (pipes)
- Обработку последовательностей команд с логическими операторами
- Управление фоновыми процессами
- Обработку ошибок и возвращаемых статусов

2 Структура модуля

2.1 Заголовочные файлы

```c
#include "shell.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <errno.h>
#include <fcntl.h>
```

3 Функции управления перенаправлениями

3.1 Функция `apply_redirections`

```c
int apply_redirections(command_t *cmd) {
    //                                              (stdin)
    if (cmd->input_file != NULL) {
        int fd = open(cmd->input_file, O_RDONLY);
        if (fd < 0) {
            perror("open input");
            return -1;
        }

        if (dup2(fd, STDIN_FILENO) < 0) {
            perror("dup2 stdin");
            close(fd);
            return -1;
        }
        close(fd);
    }

    //                                               (stdout)
    if (cmd->output_file != NULL) {
        int flags = O_WRONLY | O_CREAT;
        if (cmd->append_output) {
            flags |= O_APPEND;
        } else {
            flags |= O_TRUNC;
        }

        int fd = open(cmd->output_file, flags, 0644);
        if (fd < 0) {
            perror("open output");
            return -1;
        }

        if (dup2(fd, STDOUT_FILENO) < 0) {
            perror("dup2 stdout");
            close(fd);
            return -1;
        }
        close(fd);
    }

    //                                               (stderr)
    if (cmd->error_file != NULL) {
        int flags = O_WRONLY | O_CREAT;
        if (cmd->append_error) {
            flags |= O_APPEND;
        } else {
            flags |= O_TRUNC;
        }

        int fd = open(cmd->error_file, flags, 0644);
        if (fd < 0) {
            perror("open error");
            return -1;
        }

        if (dup2(fd, STDERR_FILENO) < 0) {
            perror("dup2 stderr");
            close(fd);
            return -1;
        }
```

```
        close(fd);
    }

    //                          stderr    stdout
    if (cmd->merge_output) {
        if (dup2(STDOUT_FILENO, STDERR_FILENO) < 0) {
            perror("dup2 merge");
            return -1;
        }
    }

    return 0;
}
```

Назначение: Применение всех перенаправлений ввода/вывода для команды.
Поддерживаемые перенаправления:

- **Ввод** (<): Открытие файла для чтения в STDIN
- **Вывод** (>): Создание/перезапись файла для STDOUT
- **Вывод с дополнением** (»): Добавление в файл для STDOUT
- **Ошибки** (2>): Создание/перезапись файла для STDERR
- **Ошибки с дополнением** (2»): Добавление в файл для STDERR
- **Объединение** (2>&1): Перенаправление STDERR в STDOUT

Возвращаемые значения:

- 0 - успешное применение перенаправлений
- -1 - ошибка при применении перенаправлений

4 Функции поиска и выполнения команд

4.1 Функция get_full_path

```
char *get_full_path(const char *command) {
    if (command == NULL || command[0] == '\0') {
        return NULL;
    }

    //

    if (command[0] == '/' || (command[0] == '.' && (command[1] == '/' ||
        (command[1] == '.' && command[2] == '/')))) {
        if (access(command, X_OK) == 0) {
            return strdup(command);
        }
        return NULL;
    }

    char *path_env = getenv("PATH");
    if (path_env == NULL) {
        return NULL;
    }
```

```
    char *path_copy = strdup(path_env);
    if (path_copy == NULL) {
        perror("strdup");
        return NULL;
    }

    char *dir = strtok(path_copy, ":");
    char *found_path = NULL;

    while (dir != NULL) {
        //
        size_t dir_len = strlen(dir);
        size_t cmd_len = strlen(command);

        if (dir_len + cmd_len + 2 > MAX_PATH_LENGTH) {
            dir = strtok(NULL, ":");
            continue;
        }

        char full_path[MAX_PATH_LENGTH];
        int written = snprintf(full_path, sizeof(full_path), "%s/%s", dir,
    command);

        if (written < 0 || written >= (int)sizeof(full_path)) {
            dir = strtok(NULL, ":");
            continue;
        }

        if (access(full_path, X_OK) == 0) {
            found_path = strdup(full_path);
            break;
        }

        dir = strtok(NULL, ":");
    }

    free(path_copy);
    return found_path;
}
```

Назначение: Поиск полного пути к исполняемому файлу команды.
Алгоритм поиска:

1. Проверка абсолютных и относительных путей
2. Поиск в директориях переменной PATH
3. Проверка прав доступа на выполнение
4. Возврат первого найденного исполняемого файла

4.2 Функция `execute_external`

```
int execute_external(command_t *cmd) {
    //
    char *full_path = get_full_path(cmd->words[0]);
    if (full_path == NULL) {
        fprintf(stderr, "%s: command not found\n", cmd->words[0]);
```

```
        return 127;
    }

    pid_t pid = fork();

    if (pid == -1) {
        perror("fork");
        free(full_path);
        return 1;
    } else if (pid == 0) {
        //

        //
        if (apply_redirections(cmd) < 0) {
            fprintf(stderr, "Error: failed to apply redirections\n");
            free(full_path);
            exit(1);
        }

        execv(full_path, cmd->words);

        //          execv                                    -

        perror("execv");
        free(full_path);
        exit(1);
    } else {
        //
        free(full_path);

        if (!cmd->fonius) {
            int status;
            waitpid(pid, &status, 0);
            return WEXITSTATUS(status);
        } else {
            printf("[%d] Started in fonius\n", pid);
            return 0;
        }
    }
}
```

Назначение: Выполнение внешней команды.
Алгоритм работы:

1. Поиск полного пути к команде
2. Создание дочернего процесса через `fork()`
3. В дочернем процессе:
   - Применение перенаправлений
   - Замена образа процесса через `execv()`
4. В родительском процессе:
   - Ожидание завершения (для foreground процессов)
   - Немедленный возврат (для background процессов)

Возвращаемые значения:

- 0-255 - код возврата команды
- 127 - команда не найдена

5 Функции работы с конвейерами

5.1 Функция split_pipeline

```
command_t **split_pipeline(command_t *cmd, int *cmd_count) {
    //

    *cmd_count = 1;
    for (int i = 0; i < cmd->word_num; i++) {
        if (strcmp(cmd->words[i], "|") == 0) {
            (*cmd_count)++;
        }
    }

    //
    command_t **commands = malloc(*cmd_count * sizeof(command_t *));
    if (commands == NULL) {
        perror("malloc");
        return NULL;
    }

    int start = 0;
    int cmd_index = 0;

    for (int i = 0; i <= cmd->word_num; i++) {
        //

        if (i == cmd->word_num || strcmp(cmd->words[i], "|") == 0) {
            //
            command_t *new_cmd = malloc(sizeof(command_t));
            if (new_cmd == NULL) {
                perror("malloc");
                //

                for (int j = 0; j < cmd_index; j++) {
                    free_command(commands[j]);
                }
                free(commands);
                return NULL;
            }
            // ...

        }
    }
    return commands;
}
```

Назначение: Разделение конвейера на отдельные команды.

5.2 Функция execute_pipeline

```
int execute_pipeline(command_t *cmd) {
    int cmd_count;
```

```
    command_t **commands = split_pipeline(cmd, &cmd_count);

    if (commands == NULL) {
        return 1;
    }

    if (cmd_count < 2) {
        fprintf(stderr, "Invalid pipeline: need at least 2 commands\n");
        for (int i = 0; i < cmd_count; i++) {
            free_command(commands[i]);
        }
        free(commands);
        return 1;
    }

    //                                pipe'
    int (*pipefds)[2] = malloc((cmd_count - 1) * sizeof(int[2]));
    if (pipefds == NULL) {
        perror("malloc");
        // ...
        return 1;
    }

    //                          pipe'
    for (int i = 0; i < cmd_count - 1; i++) {
        if (pipe(pipefds[i]) == -1) {
            perror("pipe");
            // ...                    pipe'

            return 1;
        }
    }

    //                                        PID'

    pid_t *pids = malloc(cmd_count * sizeof(pid_t));
    if (pids == NULL) {
        perror("malloc");
        // ...
        return 1;
    }

    //
    for (int i = 0; i < cmd_count; i++) {
        pids[i] = fork();

        if (pids[i] == -1) {
            perror("fork");
            //
            for (int j = 0; j < i; j++) {
                kill(pids[j], SIGTERM);
            }
            // ...
            return 1;
        }

        if (pids[i] == 0) {
            //

            //
```

```
            if (i > 0) {
                dup2(pipefds[i-1][0], STDIN_FILENO);
                close(pipefds[i-1][0]);
            }

            //
            if (i < cmd_count - 1) {
                dup2(pipefds[i][1], STDOUT_FILENO);
                close(pipefds[i][1]);
            }

            //                              pipe'

            for (int j = 0; j < cmd_count - 1; j++) {
                if (j != i - 1) close(pipefds[j][0]);
                if (j != i) close(pipefds[j][1]);
            }

            //

            if (apply_redirections(commands[i]) < 0) {
                fprintf(stderr, "Error: failed to apply redirections for
    command %d\n", i);
                exit(1);
            }

            //
            int result = execute_command(commands[i]);

            //
            for (int j = 0; j < cmd_count; j++) {
                if (j != i) free_command(commands[j]);
            }
            free(commands);
            free(pipefds);
            free(pids);

            exit(result);
        }
    }

    //                                          -
    pipe'
    for (int i = 0; i < cmd_count - 1; i++) {
        close(pipefds[i][0]);
        close(pipefds[i][1]);
    }

    //
    int last_status = 0;
    for (int i = 0; i < cmd_count; i++) {
        int status;
        waitpid(pids[i], &status, 0);
        if (i == cmd_count - 1) {  //

            last_status = WEXITSTATUS(status);
        }
    }

    //
```

```
    free(pipefds);
    free(pids);
    for (int i = 0; i < cmd_count; i++) {
        free_command(commands[i]);
    }
    free(commands);

    return last_status;
}
```

Назначение: Выполнение конвейера команд.
Алгоритм работы:

1. Разделение конвейера на отдельные команды
2. Создание необходимого количества pipe'ов
3. Создание процессов для каждой команды
4. Настройка перенаправлений между процессами:
   - Первая команда: STDIN → исходный, STDOUT → pipe1
   - Промежуточные: STDIN → pipeN-1, STDOUT → pipeN
   - Последняя команда: STDIN → pipeN-1, STDOUT → исходный
5. Ожидание завершения всех процессов
6. Возврат статуса последней команды

6 Функции работы с последовательностями команд

6.1 Функция `execute_command_sequence`

```
int execute_command_sequence(command_sequence_t *seq) {
    if (seq == NULL || seq->command_count == 0) {
        return 0;
    }

    //                                                  -

    if (seq->command_count == 1) {
        return execute_command(seq->commands[0]);
    }

    int last_status = 0;
    int should_execute_next = 1; //          :


    for (int i = 0; i < seq->command_count; i++) {
        //

        int current_status = 0;
        if (should_execute_next) {
            current_status = execute_command(seq->commands[i]);
        } else {
```

```
            //                                                  ,


            current_status = last_status;
        }

        last_status = current_status;

        //                      ,

        if (i < seq->command_count - 1) {
            int separator = seq->separators[i];

            switch (separator) {
                case 0: // ; -

                    should_execute_next = 1;
                    break;

                case 1: // && -
                                          (status == 0)
                    should_execute_next = (current_status == 0);
                    break;

                case 2: // || -
                                            (status != 0)
                    should_execute_next = (current_status != 0);
                    break;

                default:
                    fprintf(stderr, "Unknown separator type: %d\n",
    separator);
                    should_execute_next = 1;
                    break;
            }
        }
     }

     return last_status;
}
```

Назначение: Выполнение последовательности команд с учетом логических операторов.

Логика выполнения:

- **; (точка с запятой)**: Всегда выполнять следующую команду
- **&& (логическое И)**: Выполнять следующую команду только при успехе (status = 0)
- **|| (логическое ИЛИ)**: Выполнять следующую команду только при ошибке (status 0)

7 Главная функция выполнения

7.1 Функция `execute_command`

```c
int execute_command(command_t *cmd) {
    if (cmd == NULL || cmd->word_num == 0) {
        return 0;
    }

    //

    int has_separators = 0;
    for (int i = 0; i < cmd->word_num; i++) {
        if (is_command_separator(cmd->words[i])) {
            has_separators = 1;
            break;
        }
    }

    if (has_separators) {
        //

        char *reconstructed_input = malloc(MAX_INPUT_LENGTH);
        if (reconstructed_input == NULL) {
            perror("malloc");
            return 1;
        }

        reconstructed_input[0] = '\0';

        //

        for (int j = 0; j < cmd->word_num; j++) {
            if (j > 0) {
                strcat(reconstructed_input, " ");
            }
            strcat(reconstructed_input, cmd->words[j]);
        }

        //

        command_sequence_t *seq = parse_input_with_separators(
    reconstructed_input);
        free(reconstructed_input);

        if (seq != NULL) {
            int result = execute_command_sequence(seq);
            free_command_sequence(seq);
            return result;
        } else {
            return 1; //
        }
    }

    //
    for (int i = 0; i < cmd->word_num; i++) {
        if (strcmp(cmd->words[i], "|") == 0) {
            return execute_pipeline(cmd);
        }
    }

    //
    int builtin_result = execute_bash_cmd(cmd->words);
    if (builtin_result != -1) {
```

```
56          return builtin_result;
57      }
58
59      //
60      return execute_external(cmd);
61  }
```

Назначение: Главная функция выполнения команды, определяющая тип команды и направляющая ее на соответствующий обработчик.

Алгоритм определения типа команды:

1. Проверка на пустую команду
2. Проверка на наличие разделителей команд (последовательность)
3. Проверка на наличие конвейера (|)
4. Проверка на встроенную команду
5. Выполнение как внешней команды

8 Примеры выполнения

8.1 Пример 1: Простая внешняя команда

```
Ввод: "ls -la"
Действие:
1. Поиск /bin/ls через PATH
2. fork() + execv()
3. Ожидание завершения
4. Возврат статуса
```

8.2 Пример 2: Конвейер

```
Ввод: "ls -la | grep test | wc -l"
Действие:
1. Создание 2 pipe'ов
2. Запуск 3 процессов:
   - ls -la (STDOUT → pipe1)
   - grep test (STDIN ← pipe1, STDOUT → pipe2)
   - wc -l (STDIN ← pipe2)
3. Ожидание всех процессов
4. Возврат статуса wc -l
```

8.3 Пример 3: Последовательность с логическими операторами

```
Ввод: "ls /nonexistent && echo success || echo failure"
Действие:
1. Выполнение ls /nonexistent (ошибка)
2. Пропуск echo success (из-за &&)
3. Выполнение echo failure (из-за ||)
4. Возврат статуса последней команды
```

8.4 Пример 4: Команда с перенаправлениями

```
Ввод: "ls -la > output.txt 2> errors.txt"
Действие:
1. fork() создает дочерний процесс
2. В дочернем процессе:
   - STDOUT → output.txt
   - STDERR → errors.txt
   - execv(/bin/ls)
3. Родитель ожидает завершения
```

9 Особенности реализации

9.1 Управление процессами

- Корректная обработка fork() и execv()
- Ожидание завершения foreground процессов
- Отслеживание background процессов через обработчик сигналов
- Убийство процессов при ошибках создания конвейера

9.2 Управление файловыми дескрипторами

- Правильное закрытие неиспользуемых pipe'ов
- Корректное применение перенаправлений через dup2()
- Проверка ошибок открытия файлов

9.3 Обработка ошибок

- Проверка возвращаемых значений системных вызовов
- Освобождение памяти при ошибках
- Корректные коды возврата для различных ситуаций

9.4 Управление памятью

- Динамическое выделение памяти для структур команд
- Освобождение всех ресурсов при завершении
- Рекурсивное освобождение вложенных структур

10 Взаимодействие с другими модулями

10.1 Зависимости

- `shell.h` - объявления структур и констант
- `parcer.c` - предоставляет разобранные структуры команд

- `cmdfrombash.c` - обработка встроенных команд
- `main.c` - вызов функции execute_command()

10.2 Поток данных

1. `main.c` → Вызов `execute_command()`
2. `execute_command()` → Определение типа команды
3. Соответствующий обработчик → Выполнение команды
4. Возврат статуса выполнения в `main.c`

11 Заключение

Модуль `executor.c` реализует сложную и надежную систему выполнения команд, которая обеспечивает:

- Полную поддержку выполнения внешних и встроенных команд
- Реализацию конвейеров произвольной длины
- Обработку последовательностей команд с логическими операторами
- Поддержку всех типов перенаправлений ввода/вывода
- Корректное управление процессами и файловыми дескрипторами
- Надежное управление памятью и обработку ошибок

Модуль успешно интегрируется со всеми компонентами системы и обеспечивает выполнение пользовательских команд в соответствии со стандартами UNIX-подобных shell'ов.