Полная документация командного интерпретатора Shell R v7.2

Разработчик: [Ваше имя]

16 ноября 2025 г.

Аннотация

Данный документ представляет полную документацию по проекту командного интерпретатора, реализованного на языке C. Документация охватывает все основные модули системы: главный цикл выполнения, парсер командной строки, исполнитель команд, встроенные команды и главный заголовочный файл. Система обеспечивает полную функциональность UNIX-подобного shell'а с поддержкой конвейеров, перенаправлений, последовательностей команд и фонового выполнения.

Содержание

1. Введение

1.1. Обзор проекта

Командный интерпретатор Shell R v7.2 представляет собой полнофункциональную реализацию shell'a, разработанную в соответствии с требованиями практикума. Система поддерживает все основные функции современных командных оболочек, включая:

- Выполнение внешних и встроенных команд
- Поддержку конвейеров (pipes)
- Перенаправления ввода/вывода
- Последовательности команд с логическими операторами
- Фоновое выполнение процессов
- Управление переменными окружения

1.2. Архитектура системы

Система построена по модульному принципу и состоит из следующих основных компонентов:

- **main.c** - главный цикл выполнения и взаимодействие с пользователем
- **parcer.c** - лексический анализ и парсинг командной строки
- **executor.c** - выполнение команд и управление процессами
- **cmdfrombash.c** - встроенные команды shell'a
- **shell.h** - главный заголовочный файл с объявлениями

2. Модуль main.c - Главный цикл выполнения

2.1. Обзор модуля

Файл `main.c` является главным модулем командного интерпретатора, который реализует основной цикл работы программы - чтение команд, их обработку и выполнение.

2.2. Основные функции

2.2.1 Функция `check_child`

```
void check_child(int sig) {
    int status;
    pid_t pid;

    while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
        printf("[%d] Finished with status %d\n", pid, WEXITSTATUS(status));
    }
}
```

Назначение: Обработчик сигнала `SIGCHLD` для фоновых процессов.
Функциональность:

- Использует `waitpid()` с флагом `WNOHANG` для неблокирующей проверки
- Выводит информацию о завершившихся фоновых процессах
- Автоматически вызывается при завершении любого дочернего процесса

2.2.2 Функция `print_dir`

```
char* print_dir() {
    long size = pathconf(".", _PC_PATH_MAX);
    if (size == -1) {
        size = 4096;
    }

    char *cwd = malloc((size_t)size);
    if (cwd == NULL) {
        perror("malloc");
        exit(2);
    }

    cwd = getcwd(cwd, (size_t)size);
    if (cwd == NULL) {
        perror("getcwd");
        free(cwd);
        exit(2);
    }

    return cwd;
}
```

Назначение: Получение и форматирование текущей рабочей директории для приглашения.

2.2.3 Функция `read_line`

```
char *read_line(void) {
    char *line = NULL;
    size_t linesize = 0;
    char* dir = print_dir();

    printf("%s> ", dir);
    fflush(stdout);
```

```
    free(dir);

    ssize_t num_chars_read = getline(&line, &linesize, stdin);

    if (num_chars_read <= 0) {
        free(line);
        return NULL;
    }

    if (line[num_chars_read - 1] == '\n') {
        line[num_chars_read - 1] = '\0';
    }

    return line;
}
```

Назначение: Чтение командной строки от пользователя.

2.2.4 Функция `main`

```
int main(void) {
    char *input;

    signal(SIGCHLD, check_child);

    printf("Shell R v7.2\n");
    printf("Type 'exit' to quit. Or use Ctrl + D\n");

    while ((input = read_line()) != NULL) {
        if (strlen(input) == 0) {
            free(input);
            continue;
        }

        //

        command_t *cmd = parse_input(input);

        if (cmd != NULL) {
            //
            execute_command(cmd);
            free_command(cmd);
        }

        free(input);
    }

    printf("\nGoodbye!\n");
    return 0;
}
```

Назначение: Главная функция программы, реализующая основной цикл работы.

2.3. Ключевые особенности

- **Обработка сигналов**: Корректное отслеживание фоновых процессов
- **Динамическое управление памятью**: Правильное выделение и освобождение ресурсов

- **Интерактивный интерфейс**: Информативное приглашение с текущей директорией
- **Обработка ошибок**: Проверка возвращаемых значений системных вызовов

3. Модуль parcer.c - Парсер командной строки

3.1. Обзор модуля

Файл `parcer.c` является модулем парсинга командной строки, который преобразует текстовый ввод пользователя в структурированные команды с учетом специальных символов, кавычек, экранирования и перенаправлений.

3.2. Вспомогательные функции

3.2.1 Функция `is_special_char`

```c
int is_special_char(char c) {
    const char *special_chars = "@#%!&$^;:,(){}[]";
    return (strchr(special_chars, c) != NULL);
}
```

Назначение: Проверка, является ли символ специальным.

3.2.2 Функция `is_redirection_char`

```c
int is_redirection_char(char *str) {
    return (strcmp(str, ">") == 0 || strcmp(str, ">>") == 0 ||
            strcmp(str, "<") == 0 || strcmp(str, "2>") == 0 ||
            strcmp(str, "2>>") == 0 || strcmp(str, "2>&1") == 0);
}
```

Назначение: Проверка, является ли строка оператором перенаправления.

3.2.3 Функция `is_command_separator`

```c
int is_command_separator(const char *str) {
    return (strcmp(str, ";") == 0 || strcmp(str, "&&") == 0 || strcmp(str, "
    ||") == 0);
}
```

Назначение: Проверка, является ли строка разделителем команд.

3.3. Основные функции парсинга

3.3.1 Функция `parse_input`

```c
command_t *parse_input(const char *input) {
    if (input == NULL || strlen(input) == 0) {
        return NULL;
    }

    command_t *cmd = malloc(sizeof(command_t));
    // ...
```

```
    return cmd;
}
```

Назначение: Основная функция парсинга входной строки в структуру команды.
Алгоритм работы:

1. Проверка входных данных и выделение памяти
2. Инициализация структуры `command_t`
3. Посимвольный анализ с учетом экранирования и кавычек
4. Обработка фонового режима
5. Обработка перенаправлений и конвейеров

3.3.2 Функция `parse_input_with_separators`

```
command_sequence_t *parse_input_with_separators(const char *input) {
    //
    command_t *full_cmd = parse_input(input);
    // ...

}
```

Назначение: Парсинг строки с разделителями команд на последовательность.

3.4. Обработка перенаправлений

3.4.1 Функция `process_redirections_and_pipes`

```
void process_redirections_and_pipes(command_t *cmd, char **words, int *
    word_num) {
    int i = 0;
    while (i < *word_num) {
        //

        if (strcmp(words[i], ">") == 0) {
            //


        }
        // ...
    }
}
```

Назначение: Обработка и извлечение операторов перенаправления из массива слов.
Поддерживаемые операторы:

- `>` - перенаправление вывода (перезапись)
- `»` - перенаправление вывода (дополнение)
- `<` - перенаправление ввода
- `2>` - перенаправление stderr (перезапись)
- `2»` - перенаправление stderr (дополнение)
- `2>&1` - объединение stderr с stdout

3.5. Функции освобождения памяти

3.5.1 Функция `free_command`

```
void free_command(command_t *cmd) {
    if (cmd == NULL) return;
    //                                          ,

}
```

Назначение: Полное освобождение ресурсов структуры команды.

4. Модуль executor.c - Исполнитель команд

4.1. Обзор модуля

Файл `executor.c` является модулем выполнения команд, который реализует механизмы выполнения внешних команд, встроенных команд, конвейеров, последовательностей команд с учетом логических операторов и перенаправлений ввода/вывода.

4.2. Функции управления перенаправлениями

4.2.1 Функция `apply_redirections`

```
int apply_redirections(command_t *cmd) {
    //                                              (stdin)
    if (cmd->input_file != NULL) {
        int fd = open(cmd->input_file, O_RDONLY);
        // ...
    }
    // ...
    return 0;
}
```

Назначение: Применение всех перенаправлений ввода/вывода для команды.

4.3. Функции поиска и выполнения команд

4.3.1 Функция `get_full_path`

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

}
```

Назначение: Поиск полного пути к исполняемому файлу команды.

4.3.2 Функция `execute_external`

```
int execute_external(command_t *cmd) {
    char *full_path = get_full_path(cmd->words[0]);
    pid_t pid = fork();

    if (pid == 0) {
        //                                    -
                                          execv
    } else {
        //                                          -

    }
}
```

Назначение: Выполнение внешней команды.

4.4. Функции работы с конвейерами

4.4.1 Функция `execute_pipeline`

```
int execute_pipeline(command_t *cmd) {
    int cmd_count;
    command_t **commands = split_pipeline(cmd, &cmd_count);
    //                  pipe'

}
```

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
5. Ожидание завершения всех процессов

4.5. Функции работы с последовательностями команд

4.5.1 Функция `execute_command_sequence`

```
int execute_command_sequence(command_sequence_t *seq) {
    int last_status = 0;
    int should_execute_next = 1;

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

    }
    return last_status;
}
```

Назначение: Выполнение последовательности команд с учетом логических операторов.
Логика выполнения:

- ; - всегда выполнять следующую команду
- && - выполнять только при успехе (status = 0)
- || - выполнять только при ошибке (status 0)

4.6. Главная функция выполнения

4.6.1 Функция `execute_command`

```
int execute_command(command_t *cmd) {
    //                                                                  ,
                      ,
    //
}
```

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

5. Модуль cmdfrombash.c - Встроенные команды

5.1. Обзор модуля

Файл `cmdfrombash.c` содержит реализацию встроенных команд shell'a, которые выполняются непосредственно в процессе интерпретатора без создания дочерних процессов.

5.2. Функции встроенных команд

5.2.1 Функция `from_bash_cd`

```
int from_bash_cd(char **args) {
    if (args[1] == NULL) return 1;
    if (chdir(args[1]) != 0) {
        perror("cd");
        return 1;
    }
    return 0;
}
```

Назначение: Смена текущей рабочей директории.

5.2.2 Функция `from_bash_exit`

```
int from_bash_exit(char **args) {
    int exit_code = 0;
    if (args[1] != NULL) exit_code = atoi(args[1]);
    exit(exit_code);
}
```

Назначение: Завершение работы shell'a.

5.2.3 Функция from_path

```
int from_path(char **args) {
    char *path = getenv("PATH");
    if (path == NULL) printf("PATH is not set\n");
    else printf("PATH=%s\n", path);
    return 0;
}
```

Назначение: Вывод текущего значения переменной окружения PATH.

5.2.4 Функция set_path

```
int set_path(char **args) {
    if (args[1] == NULL) return 1;
    if (setenv("PATH", args[1], 1) != 0) {
        perror("setpath");
        return 1;
    }
    return 0;
}
```

Назначение: Установка нового значения переменной PATH.

5.2.5 Функция add_to_path

```
int add_to_path(char **args) {
    //                                                          PATH
}
```

Назначение: Добавление директории в начало переменной PATH.

5.2.6 Функция reset_path

```
int reset_path(char **args) {
    const char *default_path = "...";
    if (setenv("PATH", default_path, 1) != 0) {
        perror("resetpath");
        return 1;
    }
    return 0;
}
```

Назначение: Сброс переменной PATH к значению по умолчанию.

5.3. Главная функция диспетчеризации

5.3.1 Функция execute_bash_cmd

```
int execute_bash_cmd(char **args) {
    if (strcmp(args[0], "cd") == 0) return from_bash_cd(args);
    else if (strcmp(args[0], "exit") == 0) return from_bash_exit(args);
    // ...
    return -1; //
}
```

Назначение: Определение и вызов соответствующей встроенной команды.

6. Модуль shell.h - Главный заголовочный файл

6.1. Обзор файла

Файл `shell.h` является главным заголовочным файлом проекта, который содержит все объявления структур данных, констант и функций, используемых в командном интерпретаторе.

6.2. Константы и макросы

```
#define MAX_INPUT_LENGTH 4096
#define MAX_WORDS 100
#define MAX_PATH_LENGTH 1024
```

Назначение: Определение констант для ограничения размеров данных в системе.

6.3. Структуры данных

6.3.1 Структура `command_t`

```
typedef struct {
    char **words;
    int word_num;
    int fonius;
    char *input_file;
    char *output_file;
    char *error_file;
    int append_output;
    int append_error;
    int merge_output;
    char ***pipeline;
    int pipeline_count;
} command_t;
```

Назначение: Представление разобранной команды со всеми атрибутами и перенаправлениями.

6.3.2 Структура `command_sequence_t`

```
typedef struct {
    command_t **commands;
    int command_count;
    int *separators;
} command_sequence_t;
```

Назначение: Представление последовательности команд, разделенных операторами.

6.4. Прототипы функций

6.4.1 Функции парсера

```
command_t *parse_input(const char *input);
command_sequence_t *parse_input_with_separators(const char *input);
void free_command(command_t *cmd);
void free_command_sequence(command_sequence_t *seq);
```

6.4.2 Функции исполнителя

```
int execute_command(command_t *cmd);
int execute_command_sequence(command_sequence_t *seq);
int execute_bash_cmd(char **args);
int execute_pipeline(command_t *cmd);
```

6.4.3 Встроенные команды

```
int from_bash_cd(char **args);
int from_bash_exit(char **args);
int from_path(char **args);
int set_path(char **args);
int add_to_path(char **args);
int reset_path(char **args);
```

6.4.4 Функции отладки и утилиты

```
void print_command(const command_t *cmd);
void print_command_sequence(const command_sequence_t *seq);
char *read_line(void);
char *get_full_path(const char *command);
```

7. Взаимодействие модулей

7.1. Архитектурная схема

Система построена по принципу конвейера обработки данных:

1. **Ввод**: `main.c` → чтение строки от пользователя
2. **Парсинг**: `parcer.c` → преобразование в структурированную команду
3. **Диспетчеризация**: `executor.c` → определение типа команды
4. **Выполнение**: соответствующий модуль → выполнение команды
5. **Обратная связь**: вывод результатов и обновление состояния

7.2. Потоки данных

- `main.c` `parcer.c`: передача строки для парсинга
- `main.c` `executor.c`: передача команд для выполнения
- `executor.c` `cmdfrombash.c`: вызов встроенных команд
- Все модули `shell.h`: использование общих структур и функций

7.3. Управление памятью

- **Парсер** создает структуры команд и выделяет память
- **Исполнитель** использует созданные структуры без копирования
- **main.c** отвечает за освобождение ресурсов после выполнения
- Все модули следуют единой стратегии управления памятью

8. Примеры использования системы

8.1. Пример 1: Простая команда с перенаправлением

```
$ ls -la > output.txt
```

Обработка:

1. Парсинг: создание `command_t` с `words = ["ls la"]`, `output_file = "output.txt"`
2. Выполнение: `fork()` + `execv()` с применением перенаправления

8.2. Пример 2: Конвейер

```
$ ps aux | grep python | wc -l
```

Обработка:

1. Парсинг: разделение на 3 команды в конвейере
2. Выполнение: создание 2 pipe'ов и 3 процессов с соединением STDIN/STDOUT

8.3. Пример 3: Последовательность с логическими операторами

```
$ make && make test || echo "Build failed"
```

Обработка:

1. Парсинг: создание последовательности из 3 команд с разделителями && и ||
2. Выполнение: условное выполнение на основе статусов предыдущих команд

9. Заключение

9.1. Достигнутая функциональность

Разработанный командный интерпретатор обеспечивает полный набор функций, требуемых для получения высшей оценки:

- Выполнение внешних и встроенных команд
- Поддержка конвейеров произвольной длины
- Все виды перенаправлений ввода/вывода

- Последовательности команд с операторами ;, &&, ||
- Фоновое выполнение процессов
- Управление переменными окружения
- Обработка кавычек и экранирования

9.2. Архитектурные преимущества

- **Модульность**: Четкое разделение ответственности между компонентами
- **Расширяемость**: Легкое добавление новых встроенных команд и функций
- **Надежность**: Корректная обработка ошибок и управление памятью
- **Эффективность**: Минимальное копирование данных между модулями

9.3. Перспективы развития

Система может быть расширена следующими функциями:

- Поддержка скриптовых файлов
- История команд
- Автодополнение команд
- Подстановка переменных окружения
- Job control (управление заданиями)

Разработанный командный интерпретатор представляет собой законченную, надежную и расширяемую систему, соответствующую всем современным требованиям к UNIX-подобным оболочкам.