Документация к файлу shell.h
Главный заголовочный файл

Разработчик: [Ваше имя]

16 ноября 2025 г.

Аннотация

Данный документ описывает структуру и содержимое главного заголовочного файла `shell.h`, который содержит все объявления структур данных, констант и функций командного интерпретатора. Файл служит единой точкой входа для всех модулей системы.

Содержание

1 Обзор файла

Файл `shell.h` является центральным заголовочным файлом проекта, который:

- Определяет основные структуры данных для представления команд
- Объявляет константы и макросы для ограничений системы
- Содержит прототипы всех функций, используемых в проекте
- Обеспечивает согласованность между модулями
- Предоставляет интерфейс для взаимодействия компонентов системы

2 Структура файла

2.1 Защита от множественного включения

```
#ifndef SHELL_H
#define SHELL_H
```

Назначение: Предотвращение множественного включения заголовочного файла с помощью include guards.

2.2 Включаемые системные заголовочные файлы

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <signal.h>
#include <errno.h>
```

Назначение: Включение всех необходимых системных заголовочных файлов для обеспечения работы всех модулей системы.

Описание включенных файлов:

- `stdio.h` - стандартный ввод/вывод (`printf`, `perror`)
- `stdlib.h` - стандартные функции (`malloc`, `free`, `exit`)
- `string.h` - функции работы со строками (`strcmp`, `strdup`)
- `unistd.h` - POSIX API (`fork`, `exec`, `chdir`)
- `sys/types.h` - системные типы данных (`pid_t`)
- `sys/wait.h` - функции ожидания процессов (`waitpid`)
- `sys/stat.h` - работа с файловой системой
- `fcntl.h` - управление файловыми дескрипторами (`open`, `O_CREAT`)
- `signal.h` - обработка сигналов (`signal`, `SIGCHLD`)
- `errno.h` - коды ошибок

3 Константы и макросы

3.1 Ограничения системы

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

Назначение: Определение констант для ограничения размеров данных в системе.
Описание констант:

- `MAX_INPUT_LENGTH` (4096) - максимальная длина входной строки
- `MAX_WORDS` (100) - максимальное количество слов в команде
- `MAX_PATH_LENGTH` (1024) - максимальная длина пути к файлу

4 Структуры данных

4.1 Структура `command_t`

```c
typedef struct {
    char **words;
    int word_num;
    int fonius;       //
    char *input_file;    //
    char *output_file;   //                              stdout
    char *error_file;    //                              stderr
    int append_output;   //                                        stdout
    int append_error;    //                                        stderr
    int merge_output;    //                          stdout    stderr (2>&1)
    char ***pipeline;    //
    int pipeline_count;  //

} command_t;
```

Назначение: Представление разобранной команды со всеми атрибутами и перенаправлениями.

Поля структуры:

- `char **words` - массив строк (токенов команды)
- `int word_num` - количество токенов в команде
- `int fonius` - флаг фонового выполнения (1 - фоновая, 0 - foreground)
- `char *input_file` - имя файла для перенаправления ввода (<)
- `char *output_file` - имя файла для перенаправления вывода (>, »)
- `char *error_file` - имя файла для перенаправления ошибок (2>, 2»)
- `int append_output` - флаг дополнения файла вывода (»)
- `int append_error` - флаг дополнения файла ошибок (2»)
- `int merge_output` - флаг объединения stdout и stderr (2>&1)
- `char ***pipeline` - трёхмерный массив для представления конвейера команд
- `int pipeline_count` - количество команд в конвейере

4.2 Структура `command_sequence_t`

```
typedef struct {
    command_t **commands;     //
    int command_count;        //

    int *separators;          //
                        (0 - ;, 1 - &&, 2 - ||)
} command_sequence_t;
```

Назначение: Представление последовательности команд, разделенных операторами.

Поля структуры:

- `command_t **commands` - массив указателей на команды
- `int command_count` - общее количество команд в последовательности
- `int *separators` - массив типов разделителей между командами

Коды разделителей:

- 0 - точка с запятой (;) - последовательное выполнение
- 1 - логическое И (&&) - выполнение при успехе предыдущей
- 2 - логическое ИЛИ (||) - выполнение при ошибке предыдущей

5 Прототипы функций

5.1 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);
```

Назначение: Функции для разбора входных строк и управления памятью структур команд.

Описание функций:

- `parse_input()` - основной парсер командной строки
- `parse_input_with_separators()` - парсер для последовательностей команд
- `free_command()` - освобождение памяти одной команды
- `free_command_sequence()` - освобождение памяти последовательности команд

5.2 Функции исполнителя

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

Назначение: Функции для выполнения различных типов команд и управление процессами.

Описание функций:

- `execute_command()` - главная функция выполнения команды
- `execute_command_sequence()` - выполнение последовательности команд
- `execute_bash_cmd()` - диспетчер встроенных команд
- `execute_fonius()` - выполнение команд в фоновом режиме
- `execute_pipeline()` - выполнение конвейера команд

5.3 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);
```

Назначение: Функции-обработчики встроенных команд shell'а.

Описание функций:

- `from_bash_cd()` - смена текущей директории

- `from_bash_exit()` - завершение работы shell'а
- `from_path()` - вывод значения переменной PATH
- `set_path()` - установка нового значения PATH
- `add_to_path()` - добавление директории в PATH
- `reset_path()` - сброс PATH к значению по умолчанию

5.4 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 **split_line(char *line);
char *get_full_path(const char *command);

int is_command_separator(const char *str);
int get_separator_type(const char *sep);
```

Назначение: Вспомогательные функции для отладки, ввода/вывода и работы с путями.

Описание функций:

- `print_command()` - вывод отладочной информации о команде
- `print_command_sequence()` - вывод отладочной информации о последовательности
- `read_line()` - чтение строки ввода от пользователя
- `split_line()` - разделение строки на слова (альтернативная реализация)
- `get_full_path()` - поиск полного пути к исполняемому файлу
- `is_command_separator()` - проверка строки на разделитель команд
- `get_separator_type()` - получение типа разделителя

6 Завершение заголовочного файла

```
#endif
```

Назначение: Завершение блока защиты от множественного включения.

7 Архитектурные принципы

7.1 Модульность

Заголовочный файл организован таким образом, чтобы четко разделять:

- Структуры данных от функций
- Функции парсинга от функций выполнения

- Встроенные команды от внешних
- Основную функциональность от утилит

7.2 Инкапсуляция

- Все структуры данных объявлены в заголовочном файле
- Реализация функций скрыта в соответствующих .c файлах
- Четкие интерфейсы между модулями

7.3 Расширяемость

Система легко расширяема за счет:

- Добавления новых полей в структуры
- Включения новых функций с соблюдением существующей структуры
- Добавления новых констант без изменения существующего кода

8 Взаимодействие модулей

8.1 Зависимости между модулями

- `main.c` зависит от всех объявлений в `shell.h`
- `parcer.c` использует структуры `command_t` и `command_sequence_t`
- `executor.c` зависит от парсера и встроенных команд
- `cmdfrombash.c` реализует функции, объявленные в `shell.h`

8.2 Поток данных

1. `main.c` → Чтение ввода и вызов парсера
2. `parcer.c` → Создание структур команд
3. `executor.c` → Выполнение команд
4. `cmdfrombash.c` → Обработка встроенных команд

9 Примеры использования

9.1 Создание и использование команды

```c
#include "shell.h"

int example_usage() {
    //
    command_t *cmd = parse_input("ls -la > output.txt");

    if (cmd != NULL) {
        //
        print_command(cmd);

        //
        int result = execute_command(cmd);

        //
        free_command(cmd);

        return result;
    }
    return 1;
}
```

9.2 Работа с последовательностью команд

```c
#include "shell.h"

int sequence_example() {
    //
    command_sequence_t *seq = parse_input_with_separators("ls && pwd || echo
    error");

    if (seq != NULL) {
        //
        print_command_sequence(seq);

        //
        int result = execute_command_sequence(seq);

        //
        free_command_sequence(seq);

        return result;
    }
    return 1;
}
```

10 Обработка ошибок и безопасность

10.1 Проверки в функциях

Все функции, объявленные в заголовочном файле, должны:

- Проверять входные параметры на NULL
- Возвращать соответствующие коды ошибок
- Корректно обрабатывать граничные случаи

10.2 Управление памятью

- Функции создания структур должны выделять память
- Функции освобождения должны полностью очищать ресурсы
- Избегать утечек памяти при ошибках

11 Совместимость и переносимость

11.1 Стандарты соответствия

- POSIX.1-2008 для системных вызовов
- C99 для языка программирования
- Совместимость с Linux и WSL

11.2 Зависимости от системы

- Использование стандартных заголовочных файлов POSIX
- Минимальная зависимость от специфичных функций ОС
- Переносимость между различными UNIX-подобными системами

12 Заключение

Заголовочный файл `shell.h` является архитектурным ядром всего проекта командного интерпретатора. Он обеспечивает:

- **Единообразие** - все модули используют одинаковые структуры и интерфейсы
- **Модульность** - четкое разделение ответственности между компонентами
- **Расширяемость** - возможность легко добавлять новую функциональность
- **Надежность** - строгая типизация и проверки на уровне компиляции
- **Удобство сопровождения** - централизованное управление зависимостями

Благодаря хорошо продуманной структуре `shell.h`, система обладает высокой степенью связности и низким зацеплением, что делает её устойчивой к изменениям и удобной для разработки и отладки.