Лабораторные работы по курсу
Операционные системы

Лабораторная работа 2

«Основы разработки прикладных программ для ОС Linux»

2021

Оглавление

Список сокращений

ОС – Операционная Система

ПО – Программное Обеспечение

Введение

Целью лабораторной работы является освоение первичного навыка разработки прикладных программ на языке *C* для ОС *Linux*. Предполагается, что читатель уже знаком с основами языка *C* и имеет базовый опыт разработки прикладных программ (описание языка *C* приведено в официальной и последней на текущей момент спецификации *C17* [1]).

В лабораторной работе даются описание наиболее популярного инструмента в ОС *Linux* для компиляции и отладки программ «*GNU GCC Toolchain*» и основная информация о взаимодействии между прикладной программой и ОС *Linux*.

1. О языках программирования для *Linux*

В предыдущей лабораторной работе была продемонстрирована возможность разработки скриптов на языке *bash* для командной оболочки ОС *Linux*. С помощью *bash* разрабатываются, как правило, средства автоматизации взаимодействия пользователя с ОС – скрипты запуска и инициализации приложений, вспомогательные утилиты и т. д. Теоретически язык *bash* имеет в своем составе весь необходимый функционал для создания пользовательских приложений, но на практике для этих целей он применяется крайне редко по ряду причин.

Во-первых, *bash* поддерживает достаточно мало часто применяемых абстракций программирования, таких как типизация данных, структуры, классы и т. д. Их отсутствие усложняет написание и понимание кода скрипта, особенно при его большом количестве.

Во-вторых, в *bash* отсутствует пошаговая отладка исполняемого скрипта. Контроль переменных возможен только с помощью их вывода в стандартный поток (например, с помощью команды *echo*) и иных косвенных методов. Отсутствие удобного отладчика может привести к ненайденным ошибкам в логике работы ПО.

В-третьих, язык *bash* является интерпретируемым, то есть он представляет собой набор инструкций для интерпретатора, который в ходе исполнения преобразует их в инструкции для процессора, что замедляет работу скрипта и не дает возможности разработчику напрямую взаимодействовать с механизмами ОС.

Последнее время все большее распространение получили интерпретируемые языки программирования высокого уровня, такие как *Python*, *Ruby*, *JavaScript* (*Node.js*) и т. д. По отношению к *bash* они имеют большие возможности для разработки сложных приложений, поддерживают разные парадигмы и стили программирования, имеют отладчики кода. Однако ввиду интерпретируемости они также взаимодействуют с ОС и аппаратным обеспечением через программную прослойку. Это замедляет их работу и ограничивает возможности взаимодействия с ОС и аппаратным обеспечением, что может быть критично, в особенности при разработке приложений для встраиваемых систем.

Традиционно одним из часто применяемых языков для разработки приложений для встраиваемых систем является язык *C*. С одной стороны, язык *C* является языком более верхнего уровня по отношению к ассемблеру, он поддерживает основные абстракции программирования, которые позволяют создавать сложные приложения. С другой

стороны, он является компилируемым, то есть программа на *C* преобразуется в машинные инструкции до их выполнения.

Ввиду того, что ядро ОС *Linux* практически полностью было разработано на *C* – все механизмы по взаимодействию с данной ОС так же описаны именно для языка *C*. Его применение позволяет напрямую обращаться к средствам ОС и в максимально возможной степени работать с аппаратным обеспечением, что делает *C* наиболее эффективным языком для разработки высокопроизводительных приложений для встраиваемых систем.

1.1. Что такое системные вызовы

Системный вызов – это единственный механизм взаимодействия между пользовательской программой и ядром ОС *Linux*. Он позволяет программе обратиться к ОС и получить «услуги» в той части, которая ОС контролируется: запуск и остановка процессов, файловая система, интерфейсы обмена данными, аппаратное обеспечение и т. д.

Каждый системный вызов представляет собой некоторую функцию на языке *C*. Однако при вызове этой функции ее фактический код исполняется не в вызвавшей ее программе, а в системных библиотеках на уровне ядра ОС, после чего она возвращает в программу результат своей работы. Для использования системных вызовов необходимо подключение соответствующих заголовочных файлов «*.h*», в которых они объявлены.

В ОС *Linux* существует более 200 системных вызовов. Для просмотра перечня используемых приложением системных вызовов существует утилита «*strace*». После ее запуска в командной оболочке

```
strace PROG
```

на экране будут отображены все системные вызовы, которые используются программой «*PROG*».

1.2. Что такое библиотечные вызовы

Библиотечные вызовы – это стандартизированные функции языка *C*, которые «надстроены» над системными вызовами ОС: библиотечные вызовы обращаются к системным вызовам. ОС *Linux* по умолчанию использует библиотеку «*glibc*» (*GNU C Library*) для реализации библиотечных вызовов, однако она может быть заменена при необходимости. Чтобы использовать библиотечные вызовы необходимо подключение соответствующих заголовочных файлов «*.h*». В библиотечных вызовах, как правило, реализованы наиболее часто встречающиеся участки кода для работы с системными вызовами, что повышает удобство разработки.

Обращение к библиотечным вызовам несколько отвязывает программу от конкретной ОС, что делает ее более кроссплатформенной. Но при этом несколько снижается скорость работы программы и возможны некоторые потенциально опасные ситуации, связанные с параллельным исполнением программ в ОС *Linux*.

2. Инструменты разработки программ на языке C

2.1. Процесс компиляции

Процесс компиляции представляет собой процедуру перевода текстовых файлов, в которых написан исходный код программы, в машинный код, то есть последовательность инструкций, которые будут непосредственно исполняться на процессоре. Он традиционно разделяется на четыре последовательных этапа [2], показанных на Рисунке Рисунок 1:

Рисунок 1. Этапы компиляции программы

На вход компилятора подаются файлы с исходными кодами, которые могут ссылаться на функции как внутри этих файлов, так и на функции внутри внешних библиотек (в том числе на системные и библиотечные вызовы).

Первым этапом компиляции является **предобработка** исходных кодов. Она выполняется специальной подпрограммой компилятора, которая называется препроцессор. Его работа в основном связана со специальными директивами. Например, все константы, заданные в программе директивой «*#define*» будут заменены в препроцессоре на значения этой константы в тексте программы, директивы «*#include*» будут заменены текстом указанного в директиве файла и т. д.

Этап **компилирования** представляет собой преобразования исходных кодов программы из текстового формата в набор машинных инструкций на языке ассемблера. Конкретный набор используемых инструкций сильно зависит от архитектуры процессора, для которого производится компилирование. Нельзя скомпилировать исходные коды под одну архитектуру и использовать результаты компиляции на другой архитектуре.

Этап **трансляции** принимает на вход набор инструкций ассемблера и переводит его в машинный код – набор байт, которые определяют команды и операнды для процессора. Однако данный код еще невозможно исполнять на процессоре ввиду отсутствия связи между отдельными функциями и переменными.

Для определения взаимосвязей используется этап **компоновки**. Он объединяет все необходимые инструкции между собой и создает взаимосвязи между ними внутри единого исполняемого файла.

2.2. GNU C Toolchain

Компиляция исходных кодов производится с помощью специальных программ «компиляторов». В настоящее время компиляторы превратились в целые программные комплексы, которые умеют принимать на входе тексты программ на различных языках

программирования (*C*, *C++*, *Fortran*, *Go* и т. д.) и производить трансляцию и компоновку под разные процессорные архитектуры (*x86*, *ARM*, *MIPS* и т. д.).

Для разработки программ на языке *C* и их компиляции для встраиваемых систем наиболее часто используется компилятор «*GCC*» – «*GNU C Compiler*» [3]. Изначально он был разработан Ричардом Столлманом, основателем «*GNU Projects*». Он входит в состав пакета «*GNU Toolchain*», предназначенного для комплексной разработки и отладки приложений и ОС. «*GNU Toolchain*» включает в себя:

1. *GNU Compiler Collection* (*GCC*) – набор компиляторов для множества языков программирования;
2. *GNU Make* – средство автоматизации для компиляции и сборки;
3. *GNU Binutils* – набор утилит, включающих компоновщик и транслятор;
4. *GNU Debugger* (*GDB*) – средство для отладки программ;
5. *GNU Autotools* – система сборки;
6. *GNU Bison* – генератор синтаксических анализаторов.

2.2.1. Компиляция программы

В *GNU Toolchain* в ОС *Linux* частичная или полная компиляция программы осуществляется с помощью вызова программы «*gcc*». В случае, если исходный код программы состоит из одного файла, наиболее удобным способом компиляции будет вызов программы *gcc* в командной оболочке со следующими атрибутами:

```
gcc main.c -o program
```

где «*main.c*» – имя файл с исходным кодом, «*program*» – название результирующего исполняемого файла программы. Параметр ***-o*** необходим для явного указания имени исполняемого файла. Если он не указан, то исполняемый файл будет иметь название «*a.out*». Утилита *gcc* после компиляции сразу присваивает файлу программы атрибуты на исполнение.

Если программа состоит из небольшого числа файлов с исходным кодом, то ее компиляцию так же возможно производить с помощью набора команд в терминале. Для этого сначала необходимо каждый файл с исходными кодами («*.c*») преобразовать в объектный («*.o*»). Это можно сделать одной командой сразу для всех файлов:

```
gcc -c main.c file1.c file2.c
```

или же для каждого файла по отдельности:

```
gcc -c main.c
gcc -c file1.c
gcc -c file2.c
```

Вызов *gcc* для каждого файла по отдельности может быть удобен для ускорения процесса сборки программы путем преобразования в объектные только тех файлов с исходными кодами, в которых были внесены изменения относительно предыдущего этапа. Вне зависимости от выбранного способа вызова *gcc*, в рабочей директории появятся

объектные файлы «*main.o*», «*file1.o*» и «*file2.o*». Их необходимо собрать в один исполняемый файл. Это так же делается с помощью «*gcc*»:

```
gcc -c main.o file1.o file2.o -o program
```

После сборки в директории появится исполняемый файл с названием «*program*».

Если программа состоит из большого числа файлов с исходными кодами, то их сборка с помощью терминала становится крайне трудоемкой. Чтобы этого избежать в *GNU Toolchain* применяется средство автоматизации сборки, которое называется «*make*». С помощью программы *make* можно задавать очень гибкие и сложные сценарии. По сути, она представляет собой интерпретатор специального языка программирования, созданного для компиляции программ. Сами сценарии сборки описываются в файлах с помощью заранее определенного синтаксиса. Их принято называть «*make*-файлами».

По умолчанию программа *make* ищет в текущей директории файл с названием «*GNUmakefile*», «*makefile*» или «*Makefile*». Если такой файл был найден, то *make* пытается его интерпретировать как исполняемый скрипт. В рамках общей практики *make*-файл принято называть именно как «*Makefile*».

Общий внутренний формат *make*-файла выглядит следующим образом:

```
target1: зависимости
	# комментарий
правила

target2: зависимости
	правила

…

targetN: зависимости
	правила
```

Каждый указанный *target* является некоторой меткой, которой соответствует последовательность правил – набору последовательно исполняемых команд для сборки программы. Зависимости – это перечень файлов или других *target*, необходимых для выполнения данного *target*. Имя исполняемого *target* передается программе *make* в качестве аргумента. Например, при вводе команды в терминал пользователя

```
make foo
```

будет запущена программа *make*, которая будет искать в текущей директории файл с названием «*Makefile*» и выполнять последовательность правил, соответствующих *target* «*foo*».

По умолчанию каждый *target* в *Makefile* должен приводить к созданию исполняемого файла, однако часто используются так называемые «фиктивные цели» - набор правил, которые должны быть выполнены, но после которых не будет создан исполняемый файл. Они обозначаются с помощью ключевого слова «*.PHONY*». Среди таких целей наиболее часто встречаются:

- *all* – цель по умолчанию, будет вызываться без передачи конкретной цели в аргументах *make*;
- *clean* – очистка промежуточных файлов, например, объектных;

- *clean*_all – полная очистка как промежуточных так и результирующих файлов.

В целом утилита *make* обладает достаточно большими возможностями, в том числе, *make*-файл может содержать в себе переменные, производить математические операции, работать с регулярными выражениями и т.д. В рамках данного лабораторного практикума все возможности утилиты *make* рассмотрены не будут. Самостоятельно рекомендуем ознакомиться с документацией по *make* на официальном сайте [4].

2.2.2. Отладка программы

Помимо процесса компиляции программ немалый интерес для разработчика представляют средства отладки. Они позволяют отслеживать поведение программы и состояние всех внутренних переменных с помощью точек останова и в пошаговом режиме. В *GNU Toolchain* средством отладки является *GNU Debugger* (*GDB*) [5]. Он предлагает широкие возможности для исследования поведения программы, включая просмотр значений внутренних переменных и ручного вызова функций программы во время останова.

Взаимодействие между *GDB* и отлаживаемой программой может происходить как на одной машине, так и удаленно. Для этого используется специальный протокол обмена данными через последовательный порт или сеть *TCP/IP*. Для получения возможности отладки при компиляции программы должен быть задан атрибут «*-g*».

GDB не имеет собственного пользовательского графического интерфейса. Вместо этого *GDB* имеет интерфейс командной строки для интеграции во внешние *IDE* или же может быть использован в терминале. В Таблице Таблица 1 представлены некоторые команды *GDB*.

Таблица 1. Примеры команд GDB

№	Команда	Описание
1	gdb *program*	запустить отладку программы *program*
2	gdb --args program arg0 … argN	запустить отладку программы *program* с аргументами *arg0 ... argN*
3	run –v	запустить выполнение отлаживаемой программы с параметром –v
4	start	запустить программу в пошаговом режиме
5	list	вывести следующие 10 строк программы с номерами
6	list -	вывести предыдущие 10 строк программы с номерами
7	next / n	выполнить один шаг программы
8	continue / c	Запустить программу после остановки
9	info args	вывести аргументы функции
10	info locals	вывести локальные аргументы функции
11	print *var* / p *var*	вывести значение переменной *var*
12	break *function*	поставить точку останова на функции *function*
13	break N	поставить точку останова на строчке N
14	quit	выйти из GDB
15	help	полный перечень команд

Для запуска программы в режиме отладки необходимо в терминале вызвать команду *gdb* и указать ей в качестве аргумента ранее скомпилированный исполняемый файл программы:

```
gdb main
```

До вызова команды *gdb* убедитесь, что программа была скомпилирована с отладочной информацией (флаг *-g*). После вызова *gdb* в терминале отобразится интерактивный ввод команд *gdb* и программа будет доступна для запуска и отладки. По завершению исполнения программы *gdb* по-прежнему будет ожидать ввода команд. Для выхода из отладки необходимо ввести команду «*quit*».

3. Взаимодействие программ с операционной системой

В данном пункте рассматриваются базовые системные и библиотечные вызовы, которые необходимы для взаимодействия программ на языке *C* с ОС Linux.

3.1. Запуск и завершение программы

При вызове программы в командной оболочке, например,

```
./program arg0 arg1 agr2 … argN
```

в программу передаются аргументы «*arg0*», «*arg1*» и т. д. Для того, чтобы внутри кода вызываемой программы можно было корректно оперировать переданными аргументами, ее функция «*main*» должна иметь следующий вид (*Листинг 1*):

Листинг 1 – Пример функции main

```
1.  int main (int argc, char *argv[]) {
2.          int ret;
3.          // код программы
4.          return ret;
5.  }
```

где входной аргумент «*int argc*» – общее количество передаваемых аргументов, а «*char *argv[]*» – указатель на массив строк, где каждая строка содержит один из аргументов по порядку их написания («*arg0*», «*arg1*», …, «*argN*»). При этом значение *argv[0]* всегда представляет собой имя программы. Например, при вызове:

```
./program arg1 123 -f
```

значение переменной «*argc*» на входе функции «*main*» будет равно 3, а значения в массиве «*argv*»:

```
argv[0] = "./program"
argv[1] = "agr1"
argv[2] = "123"
argv[3] = "-f"
```

Каждая функция «*main*» должна иметь тип возвращаемого значения «*int*». Оно является кодом, свидетельствующем об успешности или не успешности завершения программы. Самостоятельное завершение программы возможно несколькими способами: попадание в «*return*» функции «*main*» или применение библиотечного вызова «*exit*» (или аналогичных системных вызовов) в произвольной части программы:

```
#include <stdlib.h>
void exit(int status);
```

где входной аргумент «*int status*» представляет собой возвращаемое программой значение.

По умолчанию, для большинства программ «0» в возвращаемом значении соответствует успешности выполнения, а его отрицательные значения принято воспринимать как код ошибки. Многие системные и библиотечные вызовы возвращают «-1», а конкретный код, по которому можно идентифицировать возникшую ошибку, записывают в глобальную переменную «*errno*». Для ее использования необходимо подключить соответствующий заголовочный файл:

```
#include <errno.h>
```

При завершении вызванной программы в командной оболочке возвращаемое ей значение автоматически сохраняется в переменную «?» (знак вопроса). Например, после вызова:

```
date
echo $?          # 0
```

в переменной «?» будет значение «0» (программа *date* всегда завершается успешно). Присваивать значение в переменную «?» другими способами невозможно.

3.2. Обработка строк

Во многом информационный обмен между процессом и ОС осуществляется в виде текста: входные аргументы представляют собой текстовые строки, стандартные ввод и вывод происходит в текстовом формате, основная информация о системе расположена в текстовых файлах и т. д. Стандартная библиотека языка *C* содержит порядка 20 библиотечных вызовов, упрощающих работу со строками.

Перед рассмотрением библиотечных вызовов напомним, что в языке *C* строки размещаются в массивах байт типа «*char*» или «*unsigned char*». Например,

```
char str[16];           // строка на 16 символов максимум
unsigned char[8];       // строка на 8 символов максимум
```

Обращение к строке может быть произведено путем перебора ее каждого элемента или по указателю на адрес ее начального элемента, который может быть получен разными способами:

```
char *p = &str[0];      // *p указывает на начало строки str
char *p = str;          // *p указывает на начало строки str
```

Оба присваивания «*char *p*» дают один и тот же результат.

Для возможности корректного оперирования над строкой помимо ее начального адреса необходимо так же знать либо адрес ее конечного элемента, либо ее длину. В стандартной библиотеке языка *C* конец строки определяется первым вхождением символа «*NUL*» (0x00, '\0'). Строки с таким символом в конце называют «нуль-терминированными». Именно по расположению *NUL* может быть вычислен конечный адрес строки или ее общая длина.

При инициализации строки как

```
char *s = "qwerty";
```

в конец строки уже автоматически добавляется символ *NUL*, специально добавлять его не требуется. Однако стоит понимать, что такая инициализация создает константную строку, значения в которой поменять невозможно. Например, код

```
char *s = "qwerty";
s[2] = 'a';
```

вызовет ошибку.

Рассмотрим наиболее часто используемые вызовы стандартной библиотеки *C* для работы со строками. Как правило, все они описаны в заголовочном файле «*string.h*».

Библиотечный вызов «*strlen*» принимает в качестве аргумента указатель на начало строки и возвращает ее длину в байтах без учета завершающего символа *NUL*:

```
#include <string.h>
size_t   strlen (const char *);
```

Вызовы «*strcpy*» и «*strncpy*» копируют одну строку, расположенную по адресу «*const char *src*» («*src*» получено от английского слова «*source*» – источник) в строку, расположенную по адресу «*char *dest*» («*dest*» получено от английского слова «*destination*» - место назначения):

```
#include <string.h>

char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);
```

При этом вызов «*strcpy*» копирует данные до тех пор, пока строка не закончится, а вызов «*strncpy*» копирует первые «*size_t n*» байт. Возвращаемое значение «*char **» для данных

вызовов равно адресу начала строки «*char *dest*». В Листинге 2 приведен пример использования описанных вызовов:

Листинг 2 – Пример использования функций копирования строк

```
1.  char *s = “qwerty”
2.  char d1[16];
3.  char d2[16];
4.  int len = strlen(s);    // len = 6
5.  strcpy(d1, s);         // в строку d будет скопировано «qwerty»
6.  strncpy(d2, s, 3);     // в строку d будет скопировано «qwe»
```

Библиотечные вызовы «*strcat*» и «*strncat*» добавляют одну строку к другой (конкатенируют строки):

```
#include <string.h>

char *strcat(char *dest, const char *src);
char *strncat(char *dest, const char *src, size_t n);
```

Вызов «*strcat*» добавляет к концу строки «*char *dest*» строку «*const char *src*» до тех пор, пока она не будет закончена. Вызов «*strncat*» добавляет первые «*size_t n*» символов строки «*const char *src*» к строке «*char *dest*». Например:

```
char *a = “one”;
char *b = “two”;
char d[16] = { ‘\0’ };
strcat(d, a);          // d содержит «one»
strncat(d, b, 2);      // d содержит «onetw»
```

Обратите внимание, что в приведенном примере при объявлении массива «*d*» его первый элемент инициализируется символом окончания строки. Это необходимо для того, чтобы конец строки «*d*» гарантированно находился в его нулевом элементе и вызов «*strcat*» поместило строку «*a*» в начало массива «*d*». В противном случае, при неинициализированной памяти в массиве «*d*», символ окончания строки может отсутствовать или располагаться в произвольном месте, что может привести к ошибкам.

Вызовы «*strcmp*» и «*strncmp*» сравнивают строки «*const char *s1*» и «*const char *s2*» между собой до окончания одной из них или первые «*size_t n*» байт строк соответственно:

```
#include <string.h>

int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
```

Если строки равны между собой, то вызовы «*strcmp*» и «*strncmp*» вернут «0», в противном случае число больше или меньше «0».

Одним из наиболее эффективных библиотечных вызовов, который часто применяется для обработки строк, является «*sprintf*»:

```
#include <stdio.h>
int sprintf(char *s, const char *format, ...);
```

Он позволяет сформировать и записать строку в указатель «*char *s*» по формату «*const char *format*». Возвращаемым значением «*sprintf*» является количество фактически записанных байт в «*s*», не включая символ окончания строки. Аргумент «*const char *format*» представляет собой некоторый шаблон строки, в котором специальные спецификаторы будут заменены на последующие входные аргументы вызова. Спецификатор имеет общий вид

```
%[флаги][ширина][.точность][размер]тип
```

в котором необязательные параметры определяют вид форматирования, а обязательный параметр «тип» указывает на тип преобразования соответствующего аргумента. Наиболее часто используемые спецификаторы:

- %d – десятичное знаковое число типа «int»,
- %u – десятичное беззнаковое число типа «unsigned int»,
- %.02f – число с плавающей запятой типа «double», выведенное до двух знаков после запятой,
- %s – строка,
- %c – символ.

Например, код

```
char s[32];
sprintf(s, "Hello %s! %d %.02f\n", "world", 1, 3.1415);
```

запишет в «*s*» строку «*Hello, world! 1 3.14*». Более подробно с форматами спецификаторов для «*sprintf*» можно ознакомиться в [6].

Для преобразования строки в число существуют библиотечные вызовы «*atoi*» и «*atof*». Для преобразования строки «*const char *nptr*» в целочисленную переменную типа «*int*» используется вызов «*atoi*»:

```
#include <stdlib.h>

int atoi(const char *nptr);
```

Важно понимать, что данный вызов не определяет ошибок, поэтому проверка на то, что в строке «*const char *nptr*» действительно находится целое число должна быть проведена до вызова «*atoi*».

Для преобразования строки «*const char *nptr*» в дробное число типа «*double*» применяется вызов «*atof*»:

```
#include <stdlib.h>

double atof(const char *nptr);
```

Этот вызов так же не определяет ошибок, связанных с содержанием в строке данных, отличных от дробного числа.

3.3. Выделение памяти

Строки или другие массивы могут требовать достаточно большого количество памяти для хранения данных. Объявление таких объектов как локальных переменных внутри функций определяет место их хранения в стеке, который может быть переполнен. Переполнение стека отследить достаточно трудно, однако оно практически всегда приводит к сбою работы программы. Поэтому для хранения больших объектов целесообразно использовать специальную область памяти – кучу. Для создания объектов в куче существуют системные и библиотечные вызовы. Основные из них это «*malloc*» и «*free*»:

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

Вызов «*malloc*» позволяет выделить в куче область памяти размером «*size_t size*» байт и возвращает указатель на ее начальный адрес. Вызов «*free*» высвобождает область памяти, которая начинается с адреса «*void *ptr*».

Чтобы проинициализировать каким-либо значением выделенную область памяти можно воспользоваться библиотечным вызовом «*memset*»:

```
#include <string.h>

void *memset(void *s, int c, size_t n);
```

Он заполняет первые «*size_t n*» байт памяти по указателю «*void *s*» значением «*int c*».

В Листинге 3 приведен пример использования описанных вызовов:

Листинг 3 – Пример работы со строками

```
1.   char *s = malloc(1024);
2.   memset(s, 0, sizeof(s));
3.   strcat(s, "Hello, world!\n");
4.   sprintf(s, "I'm array for %d bytes!\n", sizeof(s));
5.   …
6.   free(s);
```

С помощью «*malloc*» выделяется область памяти, доступная по указателю «*s*», а с помощью «*memset*» все элементы «*s*» устанавливаются в «0». Затем с помощью «*strcat*» и «*sprintf*» в «*s*» записываются строки. По окончанию работы с «*s*» выделенная память высвобождается.

3.4. Системные вызовы для работы с файлами и потоками

3.4.1. Работа с файлами

Обращение к файлам является одним из основных механизмов обмена информацией между прикладной программой и ОС *Linux*. Через него можно считывать и записывать информацию на диске, обращаться к виртуальным файлам с системной информацией, взаимодействовать с устройствами и т. д.

Общая последовательность действий для работы с файлом состоит из:

- открытия файла;
- чтения или записи данных;
- закрытия файла.

Операция открытия файла преобразует путь к файлу, заданному в текстовом виде, к дескриптору файла – целому числу больше «0» (тип «*int*»). Существует несколько системных вызовов, возвращающих дескриптор файла:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
```

Вызов «*open*» открывает или создает файл, а «*creat*» создает новый или перезаписывает существующий файл, если ранее он был создан.

Аргумент «*const char *pathname*» является указателем на строку, в которой содержится имя файла. Аргумент «*flags*» определяет тип доступа к файлу:

- *O_RDONLY* – только на чтение;
- *O_WRONLY* – только на запись;
- *O_RDWR* – на чтение и запись;
- *O_CREAT* – создать файл при открытии, если он не создан.
- и др.

Аргумент «*mode_t mode*» задает права доступа, которые будут применены в случае, если файл необходимо создать.

Проверку существования файла осуществляет системный вызов «*access*»:

```
#include <unistd.h>

int access(const char *pathname, int mode);
```

Его входными аргументами являются имя файла «*const char *pathname*» и режим проверки «*int mode*». Режим проверки представляет собой битовую маску, которая может состоять из одного или более флагов, объединенных побитовым «ИЛИ»:

- *R_OK* – проверка существования файла и возможности его чтения;
- *W_OK* – проверка существования файла и возможности записи в него;
- *X_OK* – проверка существования файла и возможности его исполнения;

- *F_OK* – только проверка существования файла.

В случае успеха системный вызов «*access*» возвращает «0». При ошибке, если хотя бы один из запросов «*mode*» был неудовлетворен или случались другая ошибка, вернется «-1», а код ошибки установится в переменную «*errno*».

Для получения данных из файла используется системный вызов «*read*»:

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

Первым аргументом «*int fd*» указывается дескриптор файла. Второй аргумент «*void *buf*» представляет собой указатель на начало массива, в который будут записаны данные из «*stdin*». Третий аргумент «*size_t count*» – максимальное количество считываемых байт из файла.

Системный вызов «*read*» возвращает количество фактически считанных байт. Возвращаемое значение «-1» свидетельствует об ошибке, код которой будет содержаться в «*errno*». При повторном вызове «*read*» для одного и того же дескриптора данные из файла будут последовательно считаны с того места, где остановился предыдущий вызов «*read*».

Для записи данных в файл используется системный вызов «*write*»:

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

Его первый аргумент «*int fd*» является дескриптором файла, в который будет произведена запись. Второй аргумент «*const void *buf*» – указатель на буфер, в котором расположены записываемые в поток данные. Третий аргумент «*size_t count*» – количество записываемых данных в байтах. Системный вызов «*write*» возвращает количество байт, которые были фактически записаны в поток. Возвращаемое значение «-1» свидетельствует об ошибке, код которой будет содержаться в «*errno*». Каждый вызов «*write*» для одного дескриптора файла будет последовательно дописывать данные в файл.

После завершения всех операций с файлом должен быть использован системный вызов закрытия дескриптора:

```
#include <unistd.h>

int close(int fd);
```

Он имеет только один аргумент «*int fd*» – дескриптор файла. Возвращаемое значение 0 в случае успеха или -1, если произошла ошибка.

В Листинге 4 приведен пример записи и чтения строки в файл:

Листинг 4 – Пример записи и чтения строки в файл

Следующий Листинг 5 показывает, как с помощью описанных системных вызовов записать и принять данные через последовательный порт *UART*:

Листинг 5 – Пример записи и чтения строки в файл

```
1.   // открытие последовательного порта:
2.   int fd = open("/dev/serial0", O_RDONLY | O_WRONLY | O_NOCTTY);
3.   if (fd == -1) return -1;
4.
5.   // запись данных в порт (передача):
6.   write (fd, "hello!\n", 7);
7.
8.   // чтение данных из порта (прием):
9.   char buffer[32];
10.  int n = read(fd, buffer, sizeof(buffer));
11.
12.  // закрытие последовательного порта:
13.  close (fd);
14.  return 0;
```

Из примеров видно, что стиль работы с файлом и последовательным портом между собой не различаются.

3.4.2. Работа с потоками ввода-вывода

По умолчанию для каждой вызываемой программы ОС Linux уже создает три дескриптора для стандартных потоков ввода-вывода (их описание дано в п. **Ошибка! Источник ссылки не найден.**): «*stdin*» (0), «*stdout*» (1) и «*stderr*» (2).

Запись и чтения данных из стандартных потоков ввода-вывода является частным случае работы с файлами. Отличие заключается в том, что для стандартных потоков не требуются операции открытия и закрытия, они всегда готовы для чтения и записи данных. В качестве дескрипторов стандартных потоков можно использовать их числовое обозначение или макросы:

- *STDIN_FILENO* (0) – дескриптор стандартного потока ввода *stdin*;
- *STDOUT_FILENO* (1) – дескриптор стандартного потока вывода *stdout*;
- *STDERR_FILENO* (2) – дескриптор стандартного потока вывода *stderr*;

В Листинге 6 приведен пример получения и записи данных в стандартные потоки ввода-вывода:

Листинг 6 – Пример получения и записи данных в стандартные потоки ввода-вывода

```
1.  #include <unistd.h>
2.  /* … */
3.
4.  int n;
5.  char buf[32];
6.  n = read(0, buf, sizeof(buf));
7.  n = read(STDIN_ FILENO, buf, sizeof(buf));
8.
```

```
9.  int n;
10. n = write(1, "Hello!\n", 7);
11. // write to stdout "Hello!"
12. n = write(STDOUT_FILENO, buf, sizeof(buf));
13. // write to stdout data from buf
14. n = write(2, buf, sizeof(buf));
15. // write to stderr data from buf
16. n = write(STDERR_FILENO, "Hello!\n", 7);
17. // write to stderr "Hello!"
18. /* … */
```

3.5. Библиотечные вызовы для работы с файлами и потоками

3.5.1. Работа с файлами

В библиотечных вызовах вместо дескриптора файла типа «*int*» используется понятие потока, имеющего тип «*FILE* *».

Для открытия файла используются следующие библиотечные вызовы:

```
#include <stdio.h>
FILE *fopen(const char *path, const char *mode);
FILE *fdopen(int fildes, const char *mode);
```

Вызов «*fopen*» возвращает поток «*FILE* *» для доступа к файлу, имя которого записано в строке «*const char *part*» с режимом доступа «*const char *mode*». Вызов «*fdopen*» преобразует дескриптор файла «*int filds*» в поток «*FILE* *» с режимом доступа «*const char *mode*». Возможные режимы доступа «*mode*» приведены в Таблице Таблица 2.

Таблица 2. Значения аргумента mode

Значение	**Описание**
r	Открыть файл для чтения. Чтение начинается с начала файла
r+	Открыть файл для чтения и записи. Чтение или запись начинаются с начала файла
w	«Урезать» файл до нулевой длинны или создать файл и открыть его для записи. Запись начинается с начала файла
w+	Открыть файл для чтения и записи. «Урезать» файл до нулевой длинны или создать. Чтение и запись начинаются с начала файла
A	Открыть файл для дописывания в конец. Файл создается, если не существовал
a+	Открыть для чтения и дописывания. Файл создается, если не существовал

Для закрытия потока «*FILE* *» используется библиотечный вызов «*fclose*»:

```
#include <stdio.h>
int fclose(FILE *stream);
```

При успешном закрытии возвращаемое значение «*int*» будет равно нулю. Возвращаемое значение равное «-1» свидетельствует об ошибке, код которой будет содержаться в «*errno*».

Для чтения данных из потока «*FILE **» могут использоваться следующие библиотечные вызовы:

```
#include <stdio.h>

int getc(FILE *stream);
int fgetc(FILE *stream);
char *fgets(char *s, int size, FILE *stream);
int fscanf(FILE *stream, const char *format, ...);
```

Вызовы «*getc*» и «*fgetc*» позволяют из потока «*FILE *stream*» получить один символ типа «*int*».

Вызов «*fgets*» позволяет получить из потока «*FILE *stream*» символы в количестве «*int size*» байт и записать их в строку по указателю «*char *s*».

Вызов «*fscanf*» получает из потока «*FILE *stream*» заданную в «*const char *format*» строку и передает из нее соответствующие спецификаторам символы в последующие входные аргументы вызова. Формат спецификаторов аналогичен формату спецификаторов «*sprintf*». Например, для считанной из потока «*file*» строки «*Hello 123 A*» код

```
char a[16];
int b;
char c;
fscanf(file, "%s %d %c", a, &b, &c);
```

запишет в массив «*a*» строку «*Hello*», в переменную «*b*» число «123», а в переменную «*c*» символ «*A*». Вызов «*fscanf*» возвращает значение типа «*int*», которое соответствует числу фактически преобразованных спецификаторов. Возвращаемый «0» означает, что данные были доступны, но преобразований в аргументы сделано не было. Возвращаемый символ «*EOF*» (-1) означает ошибку во время работы.

Для записи данных в поток «*FILE **» могут быть использованы следующие библиотечные вызовы:

```
#include <stdio.h>

int putc(int c, FILE *stream);
int fputc(int c, FILE *stream);
int fputs(const char *s, FILE *stream);
int fprintf(FILE *stream, const char *format, ...);
```

Вызовы «*putc*» и «*fputc*» эквивалентны между собой и записывают символ «*int c*» в поток «*FILE *stream*».

Вызов «*fputs*» записывает строку, расположенную по указателю «*const char *s*» в поток «*FILE *stream*» до тех пор, пока в строке не будет найден завершающий символ «*EOF*» (сам символ не пишется).

Вызов «*fprintf*» эквивалентен вызову «*sprintf*», но позволяет записывать данные в файловый поток «*FILE *stream*» вместо строки. Например, код

```
char a[16];
int b;
char c;
fprintf(file, "%s %d %c", "Hello", 123, A);
```

запишет в поток «*file*» строку «*Hello 123 A*».

При работе с файлами через библиотечные вызовы происходит буферизация данных. Она позволяет значительно ускорить операции чтения и записи, однако может приводить к некоторым неочевидным для разработчика проблемам: пока не сработают внутренние механизмы ОС для опустошения промежуточного буфера данные не будут реально отправлены. Чтобы принудительно отправить данные можно закрыть поток с помощью вызова «*fclose*» или использовать вызов:

```
int fflush(FILE *stream);
```

3.5.2. Работа с потоками ввода-вывода

Работа с со стандартными потоками ввода-вывода так же осуществляется вызовами, описанными в предыдущем пункте. При этом так же, как и с системными вызовами, открытие и закрытие стандартных потоков не требуется.

Для стандартных потоков в качестве потока «*FILE *»» используются следующие макросы:

- «*stdin*» - стандартный поток ввод *stdin*,
- «*stdout*» - стандартный поток вывода *stdout*,
- «*stderr*» - стандартный поток вывода *stderr*.

В библиотечных вызовах есть дополнительные специализированные вызов для чтения из стандартного потока «*stdin*»:

```
#include <stdio.h>

int getchar(void);
```

Вызов «*getchar*» эквивалентен вызову

```
int getc(stdin);
```

то есть позволяет получить один символ из стандартного потока ввода.

Для записи данных в поток так же существуют дополнительные библиотечные вызовы:

```
#include <stdio.h>

int putchar(int c);
int puts(const char *s);
int printf(const char *format, ...);
```

Вызов «*putchar*» соответствует вызову

```
putc(c, stdout)
```

Вызов «*puts*» соответствует вызову

```
fputs(s, stdout)
```

за исключением того, что «*puts*» записывает завершающий символ строки «*EOF*» в поток.

Вызов «*printf*» эквивалентен вызову «*fprintf*», только сформированная строка всегда отправляется в стандартный поток «*stdout*».

3.6. Работа с директориями

Работа с файлами зачастую сопряжена с работой с директориями: необходимо знать текущий рабочий каталог, создавать и удалять новые каталоги или знать полный путь к файлу. Для этих целей существуют как системные, так и библиотечные вызовы.

Для того, чтобы узнать текущий рабочий каталог программы можно использовать библиотечный вызов «*getcwd*»:

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);
```

Он копирует в строку «*char *buf*» размером «*size_t size*» байт текущий абсолютный (начиная от «/») путь к текущему рабочему каталогу. Возвращаемое значение «*char **» указывает на начало строки с абсолютным путем в случае успеха или на «*NULL*» в случае неуспеха, а код ошибки записывается в «*errno*».

Для смены текущего рабочего каталога можно использовать системный вызов «*chdir*»:

```
#include <unistd.h>

int chdir(const char *path);
```

Он изменяет текущий рабочий каталог на путь, указанный в «*const char *path*». В случае успеха возвращаемое значение «*int*» равно 0. В случае ошибки оно равно «-1», а код ошибки записывается в «*errno*».

Для создания новой директории с именем, указанным в строке «*const char *pathname*», и правами доступа «*mode_t mode*» можно использовать системный вызов «*mkdir*»:

```
#include <sys/stat.h>
#include <sys/types.h>

int mkdir(const char *pathname, mode_t mode);
```

В случае успеха возвращаемое значение «*int*» равно 0. В случае ошибки оно равно «-1», а код ошибки записывается в «*errno*».

Для удаления существующей директории с именем, указанным в строке «*const char *pathname*» можно использовать системный вызов «*rmdir*»:

```
#include <unistd.h>

int rmdir(const char *pathname);
```

В случае успеха возвращаемое значение «*int*» равно 0. В случае ошибки оно равно «-1», а код ошибки записывается в «*errno*».

Для получения находящихся в директории поддиректорий и файлов могут использоваться следующие библиотечные вызовы:

```
#include <sys/types.h>
#include <dirent.h>

DIR* opendir(const char* pathname);
struct dirent* readdir(DIR* dp);
int closedir(DIR* dp);
```

Вызов «*opendir*» открывает поток директории «*DIR **», имя которого находится в строке «*const char *pathname*». В случае ошибки возвращаемое значение «*DIR **» будет равно «*NULL*» и установлен код ошибки в «*errno*». После открытия потока указатель чтения находится на первом объекте (файле или поддиректории) в директории.

Вызов «*readdir*» из входного потока директории «*DIR *dp*» возвращает указатель на структуру «*struct dirent **» с информацией о найденном файле или поддиректории. При следующем вызове «*readdir*» для того же потока директории будет возвращена следующая запись. По достижении последней записи в директории будет возвращен «*NULL*». Если возникла ошибка, то «*readdir*» так же вернет «*NULL*», переменная «*errno*» изменена не будет.

Структура «*struct dirent*» может состоять из следующих полей:

```
struct dirent {
        ino_t          d_ino;                  /* Inode number */
        off_t          d_off;                  /* Not an offset; see below */
        unsigned short d_reclen;       /* Length of this record */
        unsigned char  d_type;         /* Type of file; not supported by all
filesystem types */
        char           d_name[256];    /* Null-terminated filename */
};
```

Обязательными из них в соответствии с *POSIX.1* являются только поле «*info_t d_ino*» – номер файла и поле «*char d_name[256]*» – имя файла (с символом окончания строки ‘\0’).

Вызов «*closedir*» закрывает поток «*DIR *dp*». В случае успеха возвращаемое значение «*int*» будет равно «0», а в случае ошибки оно будет равно «-1» и в «*errno*» будет установлен код ошибки.

4. Упражнения

4.1. В соответствии с п. 1 – 2 из Упражнений предыдущей лабораторной (п. **Ошибка! И сточник ссылки не найден.**) работы подключитесь к командной оболочке *Raspberry Pi*.

4.2. Перейдите в каталог с вашими проектами, создайте в нем каталог «*lab2*» для данной лабораторной работы и перейдите в него:

```
cd IVT31_Ivanov_Ivan
mkdir lab2
cd lab2
```

4.3. Создание простейшей программы на языке C

4.3.1. Создайте новый файл «*main.c*» для написания исходного кода программы и откройте его текстовым редактором «*nano*»:

```
touch main.c
nano main.c
```

4.3.2. Напишите код простейшей программы:

```
#include <stdio.h>

int main (int argc, char *argv[]) {

        printf("Hello, world!\n");

        return 0;

}
```

4.3.3. Скомпилируйте код:

```
gcc main.c - o program
```

4.3.4. Запустите программу:

```
./program
```

4.3.5. Результатам ее работы должен быть вывод строки «*Hello, world!*» на экран.

4.3.6. Добавьте исходные коды и полученную программу в систему контроля версий *git*:

```
git add main.c program
git commit -m "first simple C program"
```

4.3.7. Доработайте исходный код, добавив в него вывод входных аргументов:

4.3.8.

```
#include <stdio.h>

int main (int argc, char *argv[]) {

        printf("Hello, world!\n");

        for (int i=0; i<argc; i++) {
                printf("argc %d: %s\n", i, argv[i]);
        }

printf("The end\n");

        return 0;

}
```

4.3.9. Скомпилируйте программу и запустите:

```
gcc main.c – o program
./program abc 123 Q
```

Программа выведет первым аргументом свое название, остальные аргументы будут «*abc*», «*123*», «*Q*».

4.3.10. Добавьте изменения в *git*:

```
git add main.c program
git commit -m “Printing of arguments was added”
```

4.4. Создание автоматизированной сборки

4.4.1. Создайте *makefile* и откройте его с помощью редактора «*nano*»:

```
touch Makefile
nano Makefile
```

4.4.2. Напишите код *makefil’a*:

```
program_name = program

all:
	gcc *.c -o $(program_name)

debug:
	gcc -g *.c -o $(program_name)

clean:
	rm -f $(program_name)
	rm -f *.o
```

Данный файл имеет 3 цели сборки. Цель «*all*» компилирует все «.c» файлы в каталоге в единую программу с именем «*program*». Цель «*debug*» так же компилирует программу, но с флагом «*-g*», необходимым для отладки программы в «*gdb*». Цель «*clean*» удаляет файл программы и все промежуточные файлы компиляции.

4.4.3. Запустите очистку каталога с помощью команды

```
make clean
```

4.4.4. Запустите сборку программы с целью «*all*» с помощью команды

```
make
```

после сборки запустите программу и убедитесь в том, что она работает.

4.4.5. Обновите файл программы в *git* и добавьте туда созданный *Makefile*:

```
git add Makefile
git commit -m "Makefile was added"
```

4.5. Ознакомление с отладкой программ

4.5.1. Перекомпилируйте программу с добавлением отладочной информации:

```
make debug
```

4.5.2. Запустите программу в режиме с использованием отладчика:

```
gdb --args program abc 123 Q
```

4.5.3. Чтобы начать отладку напишите в интерактивном окне отладчика команду:

```
start
```

После ввода команды отладчик выдаст служебную информацию о старте программы и отобразит начальную строку исполняемого кода:

```
Temporary breakpoint 1, main (argc=1, argv=0xbefff654) at main.c:5
5               printf("Hello, world!\n");
```

4.5.4. Убедитесь, что данная строчка действительно имеет указанный номер строки путем просмотра кода программы с помощью ввода команды

```
list
```

Каждый ввод отображает 10 строк программы. Для возврата к предыдущим строкам необходимо вводить команду

```
list -
```

4.5.5. Просмотрите входные аргументы с помощью команды

```
info args
```

Переменная *argc* = 4, а переменная *argv* представляет собой адрес начала памяти, в которой распложены значения входных аргументов.

4.5.6. Просмотрите значения аргументов по указателю *argv* с помощью команды

```
print *argv@argc
```

Результатом работы команда будет вывод

```
$1 = {0xbefff775 "/home/pi/IVT31_Ivanov_Ivan/lab2/program", 0xbefff79d "abc", 0xbefff7a1 "123", 0xbefff7a5 "Q"}
```

Каждое нечетное значение в массиве является адресом начала аргумента в памяти, каждое четное – его значением.

4.5.7. Дважды введите команду

```
n
```

чтобы оказаться внутри цикла «*for*»

4.5.8. Введите команду

```
info locals
```

чтобы просмотреть значение локальной переменной «i».

4.5.9. С помощью команды «*list*» определите номер строчки в программе, на которой расположен

```
printf("The end\n");
```

4.5.10. Установите точку останова (брейкпоинт) на данной строчке

```
b 11
```

4.5.11. Запустите штатное выполнение программ с помощью команды

```
c
```

4.5.12. Убедитесь, что программа остановилась в точке останова.

4.5.13. Уберите точку останова с помощью команды

```
clear 11
```

4.5.14. Запустите выполнение программы до конца с помощью команды

```
c
```

4.5.15. Завершите отладку с помощью команды

```
quit
```

4.6. Разработка программы, считающей количество скрытых файлов в указанной директории

4.6.1. Измените код файла main.c на следующий:

```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <dirent.h>
#include <errno.h>


void help() {
	printf("\n");
	printf("the program for the count the number of hidden objects in directory\n\n");
	printf("Usage:\n");
	printf("	program scanning_dir\n");
	printf("	program \"\"		# for current directory\n" );
	printf("\n");
	printf("Result example:\n");
	printf("	all objects: 7\n");
	printf("	hidden objects: 1\n");
	printf("\n");
	exit(0);
}

int main (int argc, char *argv[]) {

	// help:
	if ((argc < 2) || (strcmp(argv[1], "-h") == 0)) {
		help();
	}

	// create dir path:
	char *inputdir	= argv[1];
	char *dirpath	= malloc(2048);
	dirpath[0] = 0;

	if (strlen(inputdir) == 0) {
		getcwd(dirpath, 2048);
	} else {
		strcpy(dirpath, inputdir);
	}

	// open dir stream:
	DIR *dir = opendir(dirpath);
	if (dir == NULL) {
		fprintf(stderr, "error: cannot open \"%s\" dierctory (errno code %x)\n",
dirpath, errno);
		free(dirpath);
		exit(-1);
	}
	free(dirpath);

	// scan objects in dir stream:
	struct dirent *obj = readdir(dir);
	int allobjs = 0;
	int hidobjs = 0;
	while (obj != NULL) {

		// check object name:
		if ((strcmp(obj->d_name, ".") != 0) && (strcmp(obj->d_name, "..") != 0)) {
			allobjs++;
			if (obj->d_name[0] == '.') {
				hidobjs++;
			}
		}

		obj = readdir(dir);
	}

	// send result to stdout:
	printf("all objects: %d\n", allobjs);
	printf("hidden objects: %d\n", hidobjs);
```

```
    return 0;
}
```

Вначале кода подключаются заголовочные файлы для использования библиотечных и системных вызовов. Функция «*help*» выводит справку об использовании программы.

В функции «*main*» первым блоком идет проверка на необходимость вывода справки: если аргументы не заданы или задан ключ «*-h*».

В блоке кода «*create dir path*» создается путь к директории, в которой будут обрабатываться файлы и поддиректории: если путь не задан, то берется текущий каталог, в противном случае заданный путь. Для строки с путем к директории память выделяется в куче.

В блоке кода «*open dir stream*» открывается поток директории. Если возникла ошибка, то в стандартный вывод *stderr* выводится сообщение об ошибке с указанием кода «*errno*». После использования «*dirpath*» выделенная память освобождается.

В блоке кода «*scan objects in dir stream*» перебираются все объекты, находящиеся в директории. Если именем объекта является «.» (текущая директория) или «..» (родительская директория), то он не учитывается. Общее количество объектов считается в переменной «*allobjs*», скрытые объекты («.» вначале имени) считаются в переменной «*hidobjs*».

В блоке «*send result to stdout*» результат выводится в *stdout*.

4.6.2. Скомпилируйте программу и проверьте ее работу с разными аргументами:

```
make
./program
./program -h
./program “”
./program “/home/pi”
```

Для проверки правильности выводимого результата используйте команду

```
ls -a <path>
```

4.6.3. Добавьте результаты в систему контроля версий *git*:

```
git add main.c program
git commit -m "The program for the count the hidden objects in directory was added"
```

4.7. Добавление в программу опции записи результата в файл

4.7.1. Измените код программы на следующий:

4.7.1.1. Добавьте заголовочные файлы

```
#include <sys/stat.h>
#include <fcntl.h>
```

4.7.1.2. Добавьте в функцию «*help*» пример вызова:

```
printf("        program \"\" -f file   # for write output to file\n" );
```

4.7.1.3. Измените код блока «*send result to stdout*» на следующий:

```
        // send result to stdout or file:
        char *out = malloc(128);
        sprintf (out, "all objects: %d\nhidden objects: %d\n", allobjs, hidobjs);

        if (argc < 4) {
                printf(out); // send to stdout
        } else {

                // write to file:
                if (strcmp(argv[2], "-f") != 0) {
                        free(out);
                        help();
                }
                char *outfilename = argv[3];
                int fd = open(outfilename, O_WRONLY | O_CREAT, 0777);
                if (fd == -1) {
                        fprintf(stderr, "error: cannot open \"%s\" file (errno code
%x)\n", outfilename, errno);
                        free(out);
                        exit(-1);
                }
                int n = write(fd, out, strlen(out));
                if (n == -1) {
                        close(fd);
                        fprintf(stderr, "error: cannot write data to \"%s\" file (errno
code %x)\n", outfilename, errno);
                        free(out);
                        exit(-1);
                }
                close(fd);

        }
```

Выводимое сообщение записывается в строку «*out*». Если входных аргументов меньше 4, то «*out*» выводится в «*stdout*». Если входных аргументов 4 и более, то открывается или создается файл, имя которого передано в «*argv*[3]», в него записывается «*out*» и файл закрывается.

4.7.2. Скомпилируйте и запустите программу

4.7.3.

```
make
./program “/home/pi“
./program “/home/pi“ -f tmp
```

В рабочей директории появился файл «*tmp*».

4.7.4. Выведите содержимое «*tmp*» на экран

```
cat tmp
```

и убедитесь, что оно совпадает с выводом программы в терминал.

4.7.5. Добавьте полученные результаты в *git*:

```
git add main.c program
git commit -m "optional output file instead stdout was added"
```

5. Индивидуальные задания

5.1. Общие требования к выполнению

5.1.1. Задания из п. 5.1.8 – **Ошибка! Источник ссылки не найден.** должны быть выполнены последовательно.

5.1.2. Каждое выполненное задание должно быть сохранено с использованием системы контроля версий *git*. Допускается сохранение промежуточных вариантов.

5.1.3. Коммиты в *git* должны иметь содержательное описание произведенных изменений.

5.1.4. Для каждого выполненного задания должен быть создан файл «README.MD», располагающийся в одном каталоге с исходными кодами и сохраненный с помощью git. Данный файл должен описывать поведение работы программы и показывать примеры запуска.

5.1.5. Разработанные программы должны иметь режим справки по вводимому ключу «-h» в качестве первого аргумента.

5.1.6. Конкретное задание основывается на двоичном представлении номера обучающегося в общем списке в восьмибитном формате *N*.

5.1.7. Для ознакомления принципов работы с GPIO изучите Приложение 1.

5.1.8. Продемонстрируйте преподавателю выполненное задание перед тем, как приступить к следующему.

5.2. Задание №1

Разработайте программу, которая:

5.2.1. $N_0 = 0$

просматривает указанную в аргументе 1 директорию и выводит на экран список находящихся в ней поддиректорий и файлов. С помощью опционального ключа «-f» вывод программы сохраняется в файл, имя которого указано после ключа.

5.2.2. $N_0 = 1$

просматривает указанную в аргументе 1 директорию и выводит на экран количество находящихся в ней поддиректорий и количество находящихся в ней файлов. С помощью опционального ключа «-f» вывод программы сохраняется в файл, имя которого указано после ключа.

5.3. Задание №2

5.3.1. $N_1 = 0$

выводит текстовый файл на экран построчно, добавляя перед каждой строкой ее номер. Нумерация строк начинается с единицы. Путь к файлу передается в качестве входного аргумента.

5.3.2. $N_1 = 1$

выводит статистику по текстовому файлу: общее количество строк, общее количество символов, номер и длина самой длинной строки, номер и длина самой короткой строки, средняя длинна строк. Путь к файлу передается в качестве входного аргумента.

5.4. Задание №3

Разработайте программу, которая работает с GPIO. Программа должна работать в интерактивном режиме, получая команды из стандартного потока stdin. Программа должна корректно обрабатывать следующие запросы:

SET xxxx, где x = 0 или 1 – установить соответствующий светодиод в состояние включен (1) или выключен (0);

GET – получить текущее состояние каждого светодиода;

N_{32} = 00: SHIFT L|R – сдвиг состояния светодиодов влево (L) или вправо (R);

N_{32} = 01: TOGGLE – переключение всех светодиодов на инверсное состояние;

N_{32} = 10: REVERSE – зеркальное отражение состояний светодиодов;

N_{32} = 11: SORT – изменение состояний светодиодов так, чтобы они находились в отсортированном порядке (сначала неактивные, потом активные).

Создайте bash-скрипт, перенаправляющий потоки ввода-вывода программы на файл с командами (источник) и файл, принимающий стандартный вывод программы (приемник).

6. Контрольные вопросы

1. Что такое системные и библиотечные вызовы?
2. Каковы основные этапы компиляции программы?
3. Какие инструменты используются для компиляции программы?
4. Для чего нужно выделение памяти?
5. Какие библиотечные вызовы используются для работы с файлами?

7. Список литературы

1. 9899:2017 I. C17 standard (latest draft) [Электронный ресурс] URL: https://web.archive.org/web/20181230041359if_/http://www.open-std.org/jtc1/sc22/wg14/www/abq/c17_updated_proposed_fdis.pdf
2. Ахо Альфред В. Л.М.С..С.Р.У.Д.Д. Компиляторы. Принципы, технологии и инструментарий. Вильямс, 2016.
3. GCC GNU Compiler [Электронный ресурс] URL: https://gcc.gnu.org/
4. GNU Make Manual [Электронный ресурс] URL: https://www.gnu.org/software/make/manual/
5. GDB: The GNU Project Debugger [Электронный ресурс] URL: https://www.gnu.org/software/gdb/
6. Проект OpenNet: MAN fscanf [Электронный ресурс] URL: https://www.opennet.ru/man.shtml?topic=fscanf&category=3&russian=0