

Отчет по лабораторной работе №13

Операционные системы

Дворкина Ева Владимировна

Содержание

1	Цель работы	5
2	Задание	6
3	Теоретическое введение	7
4	Выполнение лабораторной работы	9
5	Выводы	19
6	Ответы на контрольные вопросы	20

Список иллюстраций

4.1	Создание файлов и директорий	9
4.2	Функционал калькулятора в calculate.c	10
4.3	Код файла calculate.h	10
4.4	Код файла main.c	11
4.5	Компиляция программ посредством gcc	11
4.6	Makefile без исправлений	12
4.7	Makefile с исправлениями	13
4.8	Компиляция с помощью Makefile	14
4.9	Запуск GDB	14
4.10	Запуск программы	14
4.11	Постраничный просмотр кода	15
4.12	Просмотр определенных строк кода	15
4.13	Просмотр определенных строк не основного файла	15
4.14	Установка точки останова	16
4.15	Информация об имеющихся точках останова	16
4.16	Запуск программы	16
4.17	Стек вызываемых функций	16
4.18	Просмотр промежуточного значения переменной Numeral	17
4.19	Удаление точек останова	17
4.20	Анализ кода файла calculate.c	17
4.21	Анализ кода файла main.c	18

Список таблиц

1 Цель работы

Цель данной лабораторной работы - приобрести простейшие навыки разработки, анализа, тестирования и отладки приложений в ОС типа UNIX/Linux на примере создания на языке программирования С калькулятора с простейшими функциями

2 Задание

1. В домашнем каталоге создайте подкаталог `~/work/os/lab_prog`.
2. Создайте в нём файлы: `calculate.h`, `calculate.c`, `main.c`. Это будет примитивнейший калькулятор, способный складывать, вычитать, умножать и делить, возводить число в степень, брать квадратный корень, вычислять `sin`, `cos`, `tan`. При запуске он будет запрашивать первое число, операцию, второе число. После этого программа выведет результат и остановится.
3. Выполните компиляцию программы посредством `gcc`.
4. При необходимости исправьте синтаксические ошибки.
5. Создайте `Makefile` со следующим содержанием.
6. С помощью `gdb` выполните отладку программы `calcul` (перед использованием `gdb` исправьте `Makefile`)
7. С помощью утилиты `splint` попробуйте проанализировать коды файлов `calculate.c` и `main.c`.

3 Теоретическое введение

Процесс разработки программного обеспечения обычно разделяется на следующие этапы:

- планирование, включающее сбор и анализ требований к функционалу и другим характеристикам разрабатываемого приложения;
- проектирование, включающее в себя разработку базовых алгоритмов и спецификаций,
- определение языка программирования;
- непосредственная разработка приложения;
- кодирование — по сути создание исходного текста программы (возможно в нескольких вариантах);
- анализ разработанного кода;
- сборка, компиляция и разработка исполняемого модуля;
- тестирование и отладка, сохранение произведённых изменений;
- документирование.

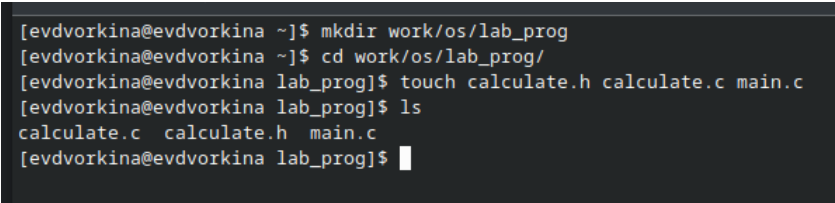
Для создания исходного текста программы разработчик может воспользоваться любым удобным для него редактором текста: vi, vim, mceditor, emacs, geany и др. После завершения написания исходного кода программы (возможно состоящей

из нескольких файлов), необходимо её скомпилировать и получить исполняемый модуль.

Стандартным средством для компиляции программ в ОС типа UNIX является GCC (GNU Compiler Collection). Это набор компиляторов для разного рода языков программирования (C, C++, Java, Фортран и др.). Работа с GCC производится при помощи одноимённой управляющей программы gcc, которая интерпретирует аргументы командной строки, определяет и осуществляет запуск нужного компилятора для входного файла. Файлы с расширением (суффиксом) .c воспринимаются gcc как программы на языке C, файлы с расширением .cc или .C — как файлы на языке C++, а файлы с расширением .o считаются объектными

4 Выполнение лабораторной работы

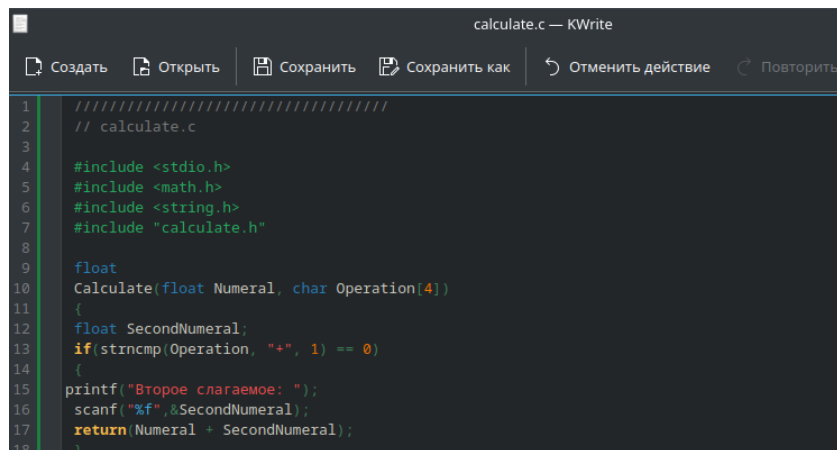
Создаю директорию `~/work/os/lab_prog`, перехожу в нее и в этой директории создаю файлы `calculate.h`, `calculate.c`, `main.c` (рис. 4.1).



```
[evdvorkina@evdvorkina ~]$ mkdir work/os/lab_prog
[evdvorkina@evdvorkina ~]$ cd work/os/lab_prog/
[evdvorkina@evdvorkina lab_prog]$ touch calculate.h calculate.c main.c
[evdvorkina@evdvorkina lab_prog]$ ls
calculate.c calculate.h main.c
[evdvorkina@evdvorkina lab_prog]$
```

Рис. 4.1: Создание файлов и директорий

Это будет примитивнейший калькулятор, способный складывать, вычитать, умножать и делить, возводить число в степень, брать квадратный корень, вычислять `sin`, `cos`, `tan`. При запуске он будет запрашивать первое число, операцию, второе число. После этого программа выведет результат и остановится. Реализация функций калькулятора в файле `calculate.h` (рис. 4.2).



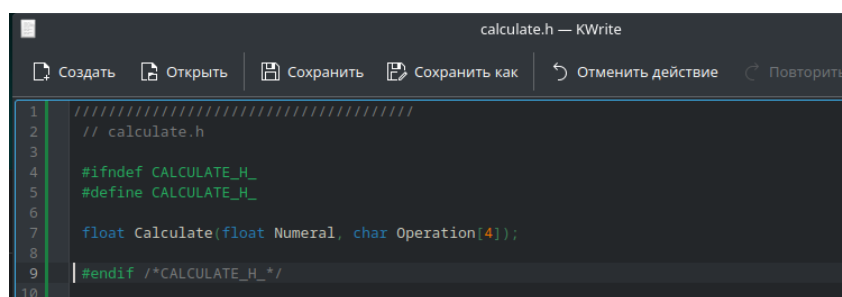
```

1 //////////////////////////////////////////////////
2 // calculate.c
3
4 #include <stdio.h>
5 #include <math.h>
6 #include <string.h>
7 #include "calculate.h"
8
9 float
10 Calculate(float Numeral, char Operation[4])
11 {
12     float SecondNumeral;
13     if(strncmp(Operation, "+", 1) == 0)
14     {
15         printf("Второе слагаемое: ");
16         scanf("%f", &SecondNumeral);
17         return Numeral + SecondNumeral;
18     }

```

Рис. 4.2: Функционал калькулятора в calculate.c

Интерфейсный файл calculate.h, описывающий формат вызова функции- калькулятора (рис. 4.3).



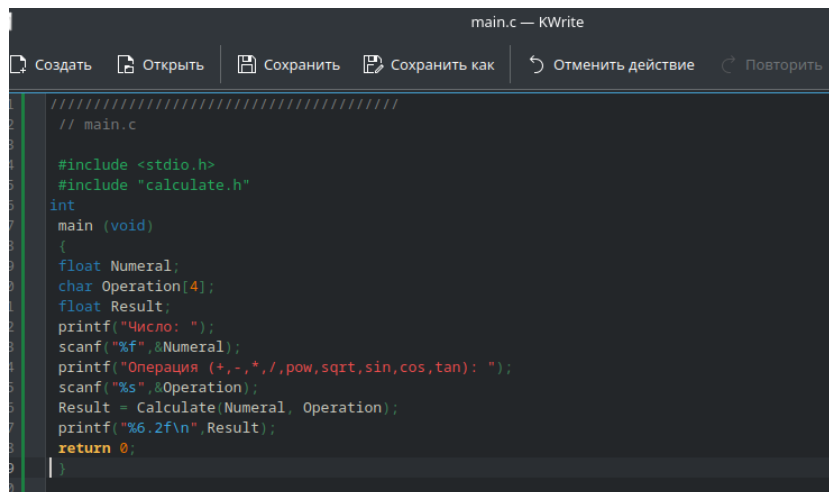
```

1 //////////////////////////////////////////////////
2 // calculate.h
3
4 #ifndef CALCULATE_H_
5 #define CALCULATE_H_
6
7 float Calculate(float Numeral, char Operation[4]);
8
9 #endif /*CALCULATE_H_*/
10

```

Рис. 4.3: Код файла calculate.h

Основной файл main.c, реализующий интерфейс пользователя к калькулятору (рис. 4.4).



```
main.c — KWrite
Создать Открыть Сохранить Сохранить как Отменить действие Повторить

// main.c

#include <stdio.h>
#include "calculate.h"
int
main (void)
{
    float Numeral;
    char Operation[4];
    float Result;
    printf("Число: ");
    scanf("%f", &Numeral);
    printf("Операция (+, -, *, /, pow, sqrt, sin, cos, tan): ");
    scanf("%s", &Operation);
    Result = Calculate(Numeral, Operation);
    printf("%.2f\n", Result);
    return 0;
}
```

Рис. 4.4: Код файла main.c

Далее выполняю компиляцию программ посредством gcc (как выяснится позже, тут не хватает флажочка) (рис. 4.5).

```
[evdvorkina@evdvorkina lab_prog]$ gcc -c calculate.c
[evdvorkina@evdvorkina lab_prog]$ gcc -c main.c
[evdvorkina@evdvorkina lab_prog]$ gcc calculate.o main.o -o calcul -lm
```

Рис. 4.5: Компиляция программ посредством gcc

Далее создаю Makefile с указанным в лабораторной работе содержанием (рис. 4.6). Но в нем нужно кое-что исправить.

```
1 #
2 # Makefile
3 #
4
5 CC = gcc
6 CFLAGS =
7 LIBS = -lm
8
9 calcul: calculate.o main.o
10 gcc calculate.o main.o -o calcul $(LIBS)
11
12 calculate.o: calculate.c calculate.h
13 gcc -c calculate.c $(CFLAGS)
14
15 main.o: main.c calculate.h
16 gcc -c main.c $(CFLAGS)
17
18 clean:
19 -rm calcul *.o *~
20
21 # End Makefile
22
```

Рис. 4.6: Makefile без исправлений

Исправляю Makefile:

Для использования GDB необходимо скомпилировать анализируемый код программы таким образом, чтобы отладочная информация содержалась в результирующем бинарном файле. Для этого следует воспользоваться опцией `-g` компилятора `gcc`

В переменную `CFLAGS` добавляю значение `-g` (рис. 4.7).

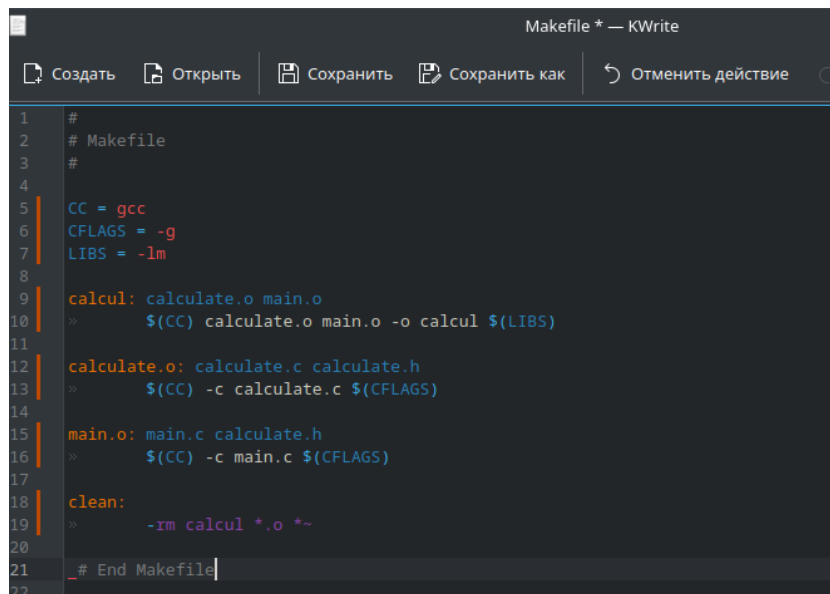


Рис. 4.7: Makefile с исправлениями

#Объявление переменных

CC = gcc #компилятор

CFLAGS = -g #опция, которая отладочную информацию положит в результирующий бинарн

LIBS = -lm #

#Создаем файл calcul из файлов calculate.o main.o

calcul: calculate.o main.o #ниже обращаемся к содержимому переменной

\$(CC) calculate.o main.o -o calcul \$(LIBS) #добавляем опцию

#Здесь отражена строка: gcc calculate.o main.o -o calcul -lm

#Создаем файл calculate.o

calculate.o: calculate.c calculate.h #

\$(CC) -c calculate.c \$(CFLAGS) #gcc -c calculate.c -g

#Создаем файл main.o

main.o: main.c calculate.h #gcc -c main.c -g

\$(CC) -c main.c \$(CFLAGS) #

#

clean: #при вызове make clean будем удалять все файлы с разрешением .o

```
-rm calcul *.o *~ #
```

Далее использую make clean, чтобы удалить не совсем верно скомпилированные файлы, и использую make (рис. 4.8).

```
[evdvorkina@evdvorkina lab_prog]$ make
gcc -c calculate.c -g
gcc -c main.c -g
gcc calculate.o main.o -o calcul -lm
```

Рис. 4.8: Компиляция с помощью Makefile

Запускаю отладчик GDB, загрузив в него программу для отладки (рис. 4.9).

```
[evdvorkina@evdvorkina lab_prog]$ gdb ./calcul
GNU gdb (GDB) Fedora Linux 12.1-7.fc37
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
  <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./calcul...
```

Рис. 4.9: Запуск GDB

Для запуска программы внутри отладчика ввожу команду run (рис. 4.10).

```
(gdb) run
Starting program: /home/evdvorkina/work/os/lab_prog/calcul

This GDB supports auto-downloading debuginfo from the following URLs:
https://debuginfod.fedoraproject.org/
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Число: 3
Операция (+, -, *, /, pow, sqrt, sin, cos, tan): +
Второе слагаемое: 2
5.00
[Inferior 1 (process 15004) exited normally]
```

Рис. 4.10: Запуск программы

Для постраничного (по 9 строк) просмотра исходного кода использую команду `list` (рис. 4.11).

```
(gdb) list
1  //////////////////////////////////////////////////
2  // main.c
3
4  #include <stdio.h>
5  #include "calculate.h"
6  int
7  main (void)
8  {
9  float Numeral;
10 char Operation[4];
(gdb)
```

Рис. 4.11: Постраничный просмотр кода

Для просмотра строк с 12 по 15 основного файла использую `list` с параметрами (рис. 4.12).

```
(gdb) list 12,15
12 printf("Число: ");
13 scanf("%f",&Numeral);
14 printf("Операция (+,-,*,/,pow,sqrt,sin,cos,tan): ");
15 scanf("%s",&Operation);
(gdb)
```

Рис. 4.12: Просмотр определенных строк кода

Для просмотра определённых строк не основного файла использую `list` с параметрами (рис. 4.13).

```
(gdb) list calculate.c:20,29
20 {
21 printf("Вычитаемое: ");
22 scanf("%f",&SecondNumeral);
23 return(Numeral - SecondNumeral);
24 }
25 else if(strncmp(Operation, "*", 1) == 0)
26 {
27 printf("Множитель: ");
28 scanf("%f",&SecondNumeral);
29 return(Numeral * SecondNumeral);
(gdb)
```

Рис. 4.13: Просмотр определенных строк не основного файла

Устанавливаю точку остановки в файле `calculate.c` на строке номер 21 (`break 21`) (рис. 4.14).

```
(gdb) list calculate.c:20,27
20
21     printf("Вычитаемое: ");
22     scanf("%f",&SecondNumeral);
23     return(Numeral - SecondNumeral);
24 }
25 else if(strncmp(Operation, "*", 1) == 0)
26 {
27     printf("Множитель: ");
(gdb) break 21
Breakpoint 1 at 0x40120f: file calculate.c, line 21.
(gdb)
```

Рис. 4.14: Установка точки остановки

Вывожу информацию об имеющихся в проекте точках остановки (рис. 4.15).

```
(gdb) info breakpoints
Num   Type      Disp Enb Address          What
1     breakpoint keep y   0x000000000040120f in Calculate at calculate.c:21
(gdb)
```

Рис. 4.15: Информация об имеющихся точках остановки

Запускаю программу внутри отладчика и убеждаюсь, что программа останавливается в момент прохождения точки остановки: (рис. 4.16).

```
(gdb) run
Starting program: /home/evdvorkina/work/os/lab_prog/calcul
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Число: 5
Операция (+, -, *, /, pow, sqrt, sin, cos, tan): -
Breakpoint 1, Calculate (Numeral=5, Operation=0x7fffffff934 "-") at calculate.c:21
21     printf("Вычитаемое: ");
(gdb)
```

Рис. 4.16: Запуск программы

Команда backtrace покажет весь стек вызываемых функций от начала программы до текущего места (рис. 4.17).

```
(gdb) backtrace
#0 Calculate (Numeral=5, Operation=0x7fffffff934 "-") at calculate.c:21
#1 0x00000000004014eb in main () at main.c:16
(gdb)
```

Рис. 4.17: Стек вызываемых функций

Смотрю, чему равно на этом этапе значение переменной Numeral, это можно сделать двумя способами. При первом я получу вывод значения переменной из

bash-скрипта, второй способ более интуитивный, там значение соответствует переменной из кода на Си (рис. 4.18).

```
(gdb) print Numeral
$1 = 5
(gdb) display Numeral
1: Numeral = 5
(gdb)
```

Рис. 4.18: Просмотр промежуточного значения переменной Numeral

Удаляю точки останова (рис. 4.19).

```
(gdb) info breakpoints
Num   Type      Disp Enb Address          What
1      breakpoint keep y  0x000000000040120f in Calculate at calculate.c:21
      breakpoint already hit 1 time
(gdb) delete 1
(gdb) info breakpoints
No breakpoints or watchpoints.
(gdb)
```

Рис. 4.19: Удаление точек останова

С помощью утилиты splint пробую проанализировать код файла calculate.c. (рис. 4.20).

```
[evdvorkina@evdvorkina lab_prog]$ splint calculate.c
Splint 3.1.2 --- 23 Jul 2022

calculate.h:7:38: Function parameter Operation declared as manifest array (size
      constant is meaningless)
      A formal parameter is declared as an array with size. The size of the array
      is ignored in this context, since the array formal parameter is treated as a
      pointer. (Use -fixedformalarray to inhibit warning)
calculate.c:10:31: Function parameter Operation declared as manifest array
      (size constant is meaningless)
calculate.c: (in function Calculate)
calculate.c:16:2: Return value (type int) ignored: scanf("%f", &Sec...
      Result returned by function call is not used. If this is intended, can cast
      result to (void) to eliminate message. (Use -retvalint to inhibit warning)
calculate.c:22:2: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:28:2: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:34:2: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:35:5: Dangerous equality comparison involving float types:
      SecondNumeral == 0
      Two real (float, double, or long double) values are compared directly using
      == or != primitive. This may produce unexpected results since floating point
      representations are inexact. Instead, compare the difference to FLT_EPSILON
      or DBL_EPSILON. (Use -realcompare to inhibit warning)
calculate.c:38:8: Return value type double does not match declared type float:
      (HUGE_VAL)
      To allow all numeric types to match, use +relaxtypes.
calculate.c:46:2: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:47:8: Return value type double does not match declared type float:
      (pow(Numeral, SecondNumeral))
calculate.c:50:8: Return value type double does not match declared type float:
      (sqrt(Numeral))
calculate.c:52:8: Return value type double does not match declared type float:
      (sin(Numeral))
calculate.c:54:8: Return value type double does not match declared type float:
      (cos(Numeral))
calculate.c:56:8: Return value type double does not match declared type float:
      (tan(Numeral))
calculate.c:60:8: Return value type double does not match declared type float:
      (HUGE_VAL)
Finished checking --- 15 code warnings
```

Рис. 4.20: Анализ кода файла calculate.c

С помощью утилиты splint пробую проанализировать код файла main.c. (рис. 4.21).

```
[evdvozkina@evdvozkina lab_prog]$ splint main.c
Splint 3.1.2 --- 23 Jul 2022

calculate.h:7:38: Function parameter Operation declared as manifest array (size
                constant is meaningless)
    A formal parameter is declared as an array with size.  The size of the array
    is ignored in this context, since the array formal parameter is treated as a
    pointer. (Use -fixedformalarray to inhibit warning)
main.c: (in function main)
main.c:13:2: Return value (type int) ignored: scanf("%f", &Num...
    Result returned by function call is not used. If this is intended, can cast
    result to (void) to eliminate message. (Use -retvalint to inhibit warning)
main.c:15:13: Format argument 1 to scanf (%s) expects char * gets char [4] *:
                &Operation
    Type of parameter is not consistent with corresponding code in format string.
    (Use -formattype to inhibit warning)
    main.c:15:10: Corresponding format code
main.c:15:2: Return value (type int) ignored: scanf("%s", &Ope...
Finished checking --- 4 code warnings
```

Рис. 4.21: Анализ кода файла main.c

5 Выводы

При выполнении данной лабораторной работы я приобрела простейшие навыки разработки, анализа, тестирования и отладки приложений в ОС типа UNIX/Linux на примере создания на языке программирования С калькулятора с простейшими функциями

6 Ответы на контрольные вопросы

1. Как получить информацию о возможностях программ gcc, make, gdb и др.?
Дополнительную информацию об этих программах можно получить с помощью функций info и man.
2. Назовите и дайте краткую характеристику основным этапам разработки приложений в UNIX. Unix поддерживает следующие основные этапы разработки приложений:
 - создание исходного кода программы;
 - представляется в виде файла;
 - сохранение различных вариантов исходного текста;
 - анализ исходного текста; Необходимо отслеживать изменения исходного кода, а также при работе более двух программистов над проектом программы нужно, чтобы они не делали изменений кода в одно время.
 - компиляция исходного текста и построение исполняемого модуля;
 - тестирование и отладка;
 - проверка кода на наличие ошибок
 - сохранение всех изменений, выполняемых при тестировании и отладке.
3. Что такое суффикс в контексте языка программирования? Приведите примеры использования. Использование суффикса “.c” для имени файла с программой на языке Си отражает удобное и полезное соглашение, принятое в ОС UNIX. Для любого имени входного файла суффикс определяет какая

компиляция требуется. Суффиксы и префиксы указывают тип объекта. Одно из полезных свойств компилятора Си — его способность по суффиксам определять типы файлов. По суффиксу .c компилятор распознает, что файл abcd.c должен компилироваться, а по суффиксу .o, что файл abcd.o является объектным модулем и для получения исполняемой программы необходимо выполнить редактирование связей. Простейший пример командной строки для компиляции программы abcd.c и построения исполняемого модуля abcd имеет вид: `gcc -o abcd abcd.c`. Некоторые проекты предпочитают показывать префиксы в начале текста изменений для старых (old) и новых (new) файлов. Опция `-prefix` может быть использована для установки такого префикса. Плюс к этому команда `bzr diff -p1` выводит префиксы в форме которая подходит для команды `patch -p1`.

4. Каково основное назначение компилятора языка C в UNIX? Основное назначение компилятора с языка Си заключается в компиляции всей программы в целом и получении исполняемого модуля.
5. Для чего предназначена утилита make? При разработке большой программы, состоящей из нескольких исходных файлов заголовков, приходится постоянно следить за файлами, которые требуют перекомпиляции после внесения изменений. Программа make освобождает пользователя от такой рутинной работы и служит для документирования взаимосвязей между файлами. Описание взаимосвязей и соответствующих действий хранится в так называемом make-файле, который по умолчанию имеет имя makefile или Makefile.
6. Приведите пример структуры Makefile. Дайте характеристику основным элементам этого файла. makefile для программы abcd.c мог бы иметь вид:

Makefile

CC = gcc

```
CFLAGS =
```

```
LIBS = -lm
```

```
calcul: calculate.o main.o
```

```
gcc calculate.o main.o -o calcul $(LIBS)
```

```
calculate.o: calculate.c calculate.h
```

```
gcc -c calculate.c $(CFLAGS)
```

```
main.o: main.c calculate.h
```

```
gcc -c main.c $(CFLAGS)
```

```
clean: -rm calcul *.o *
```

```
End Makefile
```

В общем случае make-файл содержит последовательность записей (строк), определяющих зависимости между файлами. Первая строка записи представляет собой список целевых (зависимых) файлов, разделенных пробелами, за которыми следует двоеточие и список файлов, от которых зависят целевые. Текст, следующий за точкой с запятой, и все последующие строки, начинающиеся с литеры табуляции, являются командами ОС UNIX, которые необходимо выполнить для обновления целевого файла. Таким образом, спецификация взаимосвязей имеет формат: `target1 [target2...]: [:][dependment1...] [(tab)commands] [#commentary]` `[(tab)commands] [#commentary]`, где `#` — специфицирует начало комментария, так

как содержимое строки, начиная с # и до конца строки, не будет обрабатываться командой `make`; `:` — последовательность команд ОС UNIX должна содержаться в одной строке `make`-файла (файла описаний), есть возможность переноса команд `()`, но она считается как одна строка; `::` — последовательность команд ОС UNIX может содержаться в нескольких последовательных строках файла описаний. Приведённый выше `make`-файл для программы `abcd.c` включает два способа компиляции и построения исполняемого модуля. Первый способ предусматривает обычную компиляцию с построением исполняемого модуля с именем `abcd`. Вторым способом позволяет включать в исполняемый модуль `testabcd` возможность выполнить процесс отладки на уровне исходного текста.

7. Назовите основное свойство, присущее всем программам отладки. Что необходимо сделать, чтобы его можно было использовать? Пошаговая отладка программ заключается в том, что выполняется один оператор программы и, затем контролируются те переменные, на которые должен был воздействовать данный оператор. Если в программе имеются уже отлаженные подпрограммы, то подпрограмму можно рассматривать, как один оператор программы и воспользоваться вторым способом отладки программ. Если в программе существует достаточно большой участок программы, уже отлаженный ранее, то его можно выполнить, не контролируя переменные, на которые он воздействует. Использование точек останова позволяет пропускать уже отлаженную часть программы. Точка останова устанавливается в местах, где необходимо проверить содержимое переменных или просто проконтролировать, передаётся ли управление данному оператору. Практически во всех отладчиках поддерживается это свойство (а также выполнение программы до курсора и выход из подпрограммы). Затем отладка программы продолжается в пошаговом режиме с контролем локальных и глобальных переменных, а также внутренних регистров микроконтроллера и напряжений на выводах этой микросхемы.
8. Назовите и дайте основную характеристику основным командам отладчика

`gdb.`

`backtrace` – выводит весь путь к текущей точке останова, то есть названия всех функций, начиная от `main()`; иными словами, выводит весь стек функций;

`break` – устанавливает точку останова; параметром может быть номер строки или название функции;

`clear` – удаляет все точки останова на текущем уровне стека (то есть в текущей функции);

`continue` – продолжает выполнение программы от текущей точки до конца;

`delete` – удаляет точку останова или контрольное выражение;

`display` – добавляет выражение в список выражений, значения которых отображаются каждый раз при остановке программы;

`finish` – выполняет программу до выхода из текущей функции; отображает возвращаемое значение, если такое имеется;

`info breakpoints` – выводит список всех имеющихся точек останова;

`info watchpoints` – выводит список всех имеющихся контрольных выражений;

`splist` – выводит исходный код; в качестве параметра передаются название файла исходного кода, затем, через двоеточие, номер начальной и конечной строки;

`next` – пошаговое выполнение программы, но, в отличие от команды `step`, не выполняет пошагово вызываемые функции;

`print` – выводит значение какого-либо выражения (выражение передаётся в качестве параметра);

`run` – запускает программу на выполнение;

`set` – устанавливает новое значение переменной

`step` – пошаговое выполнение программы;

`watch` – устанавливает контрольное выражение, программа остановится, как только значение контрольного выражения изменится;

9. Опишите по шагам схему отладки программы, которую Вы использовали при выполнении лабораторной работы. Выполните компиляцию про-

граммы 2) Увидели ошибки в программе Открыли редактор и исправили программу Загрузили программу в отладчик `gdb run` — отладчик выполнил программу, мы ввели требуемые значения. программа завершена, `gdb` не видит ошибок.

10. Прокомментируйте реакцию компилятора на синтаксические ошибки в программе при его первом запуске. Отладчику не понравился формат `%s` для `&Operation`, т.к `%s` — символьный формат, а значит необходим только `Operation`.
11. Назовите основные средства, повышающие понимание исходного кода программы. Если вы работаете с исходным кодом, который не вами разрабатывался, то назначение различных конструкций может быть не совсем понятным. Система разработки приложений UNIX предоставляет различные средства, повышающие понимание исходного кода. К ним относятся:

`cscope` - исследование функций, содержащихся в программе;

`splint` — критическая проверка программ, написанных на языке Си.

12. Каковы основные задачи, решаемые программой `splint`?

Проверка корректности задания аргументов всех исполняемых функций , а также типов возвращаемых ими значений;

Поиск фрагментов исходного текста, корректных с точки зрения синтаксиса языка Си, но малоэффективных с точки зрения их реализации или содержащих в себе семантические ошибки;

Общая оценка мобильности пользовательской программы.