

Отчёт по лабораторной работе № 13

дисциплина: Операционные системы

Андрианова Марина Георгиевна

Цель 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.

Выполнение лабораторной работы

1). В домашнем каталоге создадим подкаталог `~/work/os/lab_prog`(команда “`mkdir ~/work/os/lab_prog`”)(рис.1).

```
[mgandrianova@fedora ~]$ mkdir ~/work/os/lab_prog
```

Рис.1: Создание подкаталога

2). Перейдём в созданный подкаталог(команда “`cd ~/work/os/lab_prog`”) и создадим в нём файлы: `calculate.h`, `calculate.c`, `main.c`(команда “`touch calculate.h calculate.c main.c`”)(рис.2). Проверим созданные файлы командой “`ls`”(рис.3).

```
[mgandrianova@fedora ~]$ cd ~/work/os/lab_prog
[mgandrianova@fedora lab_prog]$ touch calculate.h calculate.c main.c
```

Рис.2: Создание файлов

```
[mgandrianova@fedora lab_prog]$ ls
calculate.c calculate.h main.c
```

Рис.3: Проверяем созданные файлы

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.

Запустила редактор Emacs в фоновом режиме(команда “`emacs &`”) и приступила к редактированию созданных файлов. Реализация функций калькулятора в файле `calculate.c`(рис.4):

```

////////////////////////////////////
// calculate.c

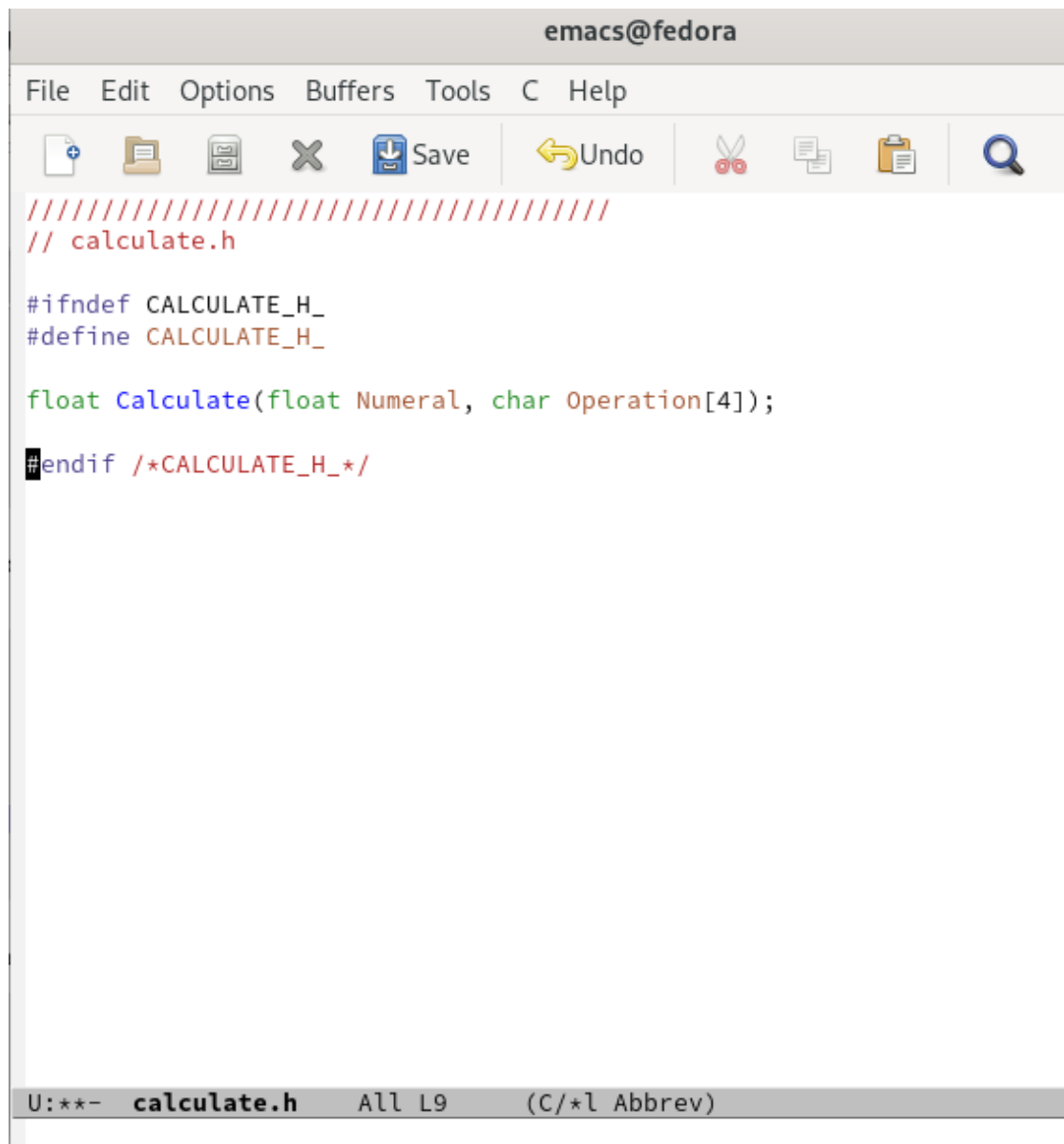
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "calculate.h"

float
Calculate(float Numeral, char Operation[4])
{
    float SecondNumeral;
    if(strncmp(Operation, "+", 1) == 0)
    {
        printf("Второе слагаемое: ");
        scanf("%f",&SecondNumeral);
        return(Numeral + SecondNumeral);
    }
    else if(strncmp(Operation, "-", 1) == 0)
    {
        printf("Вычитаемое: ");
        scanf("%f",&SecondNumeral);
        return(Numeral - SecondNumeral);
    }
    else if(strncmp(Operation, "*", 1) == 0)
    {
        printf("Множитель: ");
        scanf("%f",&SecondNumeral);
        scanf("%f",&SecondNumeral);
        return(Numeral * SecondNumeral);
    }
    else if(strncmp(Operation, "/", 1) == 0)
    {
        printf("Делитель: ");
        scanf("%f",&SecondNumeral);
        if(SecondNumeral == 0)
        {
            printf("Ошибка: деление на ноль! ");
            return(HUGE_VAL);
        }
        else
            return(Numeral / SecondNumeral);
    }
    else if(strncmp(Operation, "pow", 3) == 0)
    {
        printf("Степень: ");
        scanf("%f",&SecondNumeral);
        return(pow(Numeral, SecondNumeral));
    }
    else if(strncmp(Operation, "sqrt", 4) == 0)
        return(sqrt(Numeral));
    else if(strncmp(Operation, "sin", 3) == 0)
        return(sin(Numeral));
    else if(strncmp(Operation, "cos", 3) == 0)
        return(cos(Numeral));
    else if(strncmp(Operation, "tan", 3) == 0)
        return(tan(Numeral));
    else
    {
        printf("Неправильно введено действие ");
        return(HUGE_VAL);
    }
}

```

Рис.4: Программа в calculate.c

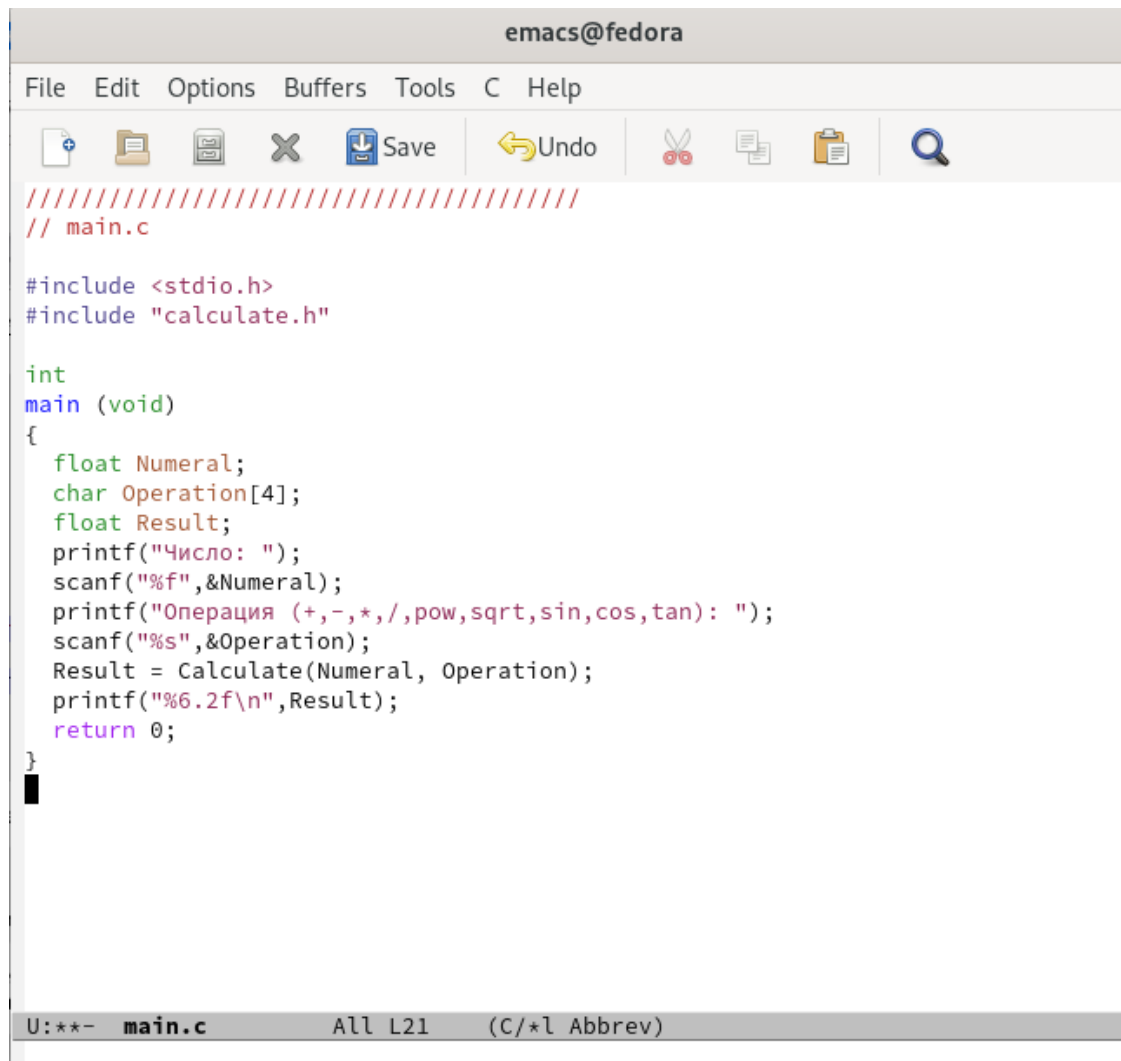
Интерфейсный файл calculate.h, описывающий формат вызова функции калькулятора(рис.5):



```
////////////////////  
// calculate.h  
  
#ifndef CALCULATE_H_  
#define CALCULATE_H_  
  
float Calculate(float Numeral, char Operation[4]);  
#endif /*CALCULATE_H_*/
```

Рис.5: Программа в calculate.h

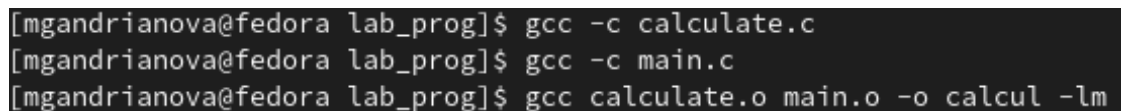
Основной файл main.c, реализующий интерфейс пользователя к калькулятору(рис.6):



```
////////////////////  
// 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("%6.2f\n",Result);  
    return 0;  
}
```

Рис.6: Программа в main.c

3). Выполним компиляцию программы посредством gcc, используя команды “gcc -c calculate.c”, “gcc -c main.c” и “gcc calculate.o main.o -o calcul -lm”(рис.7).

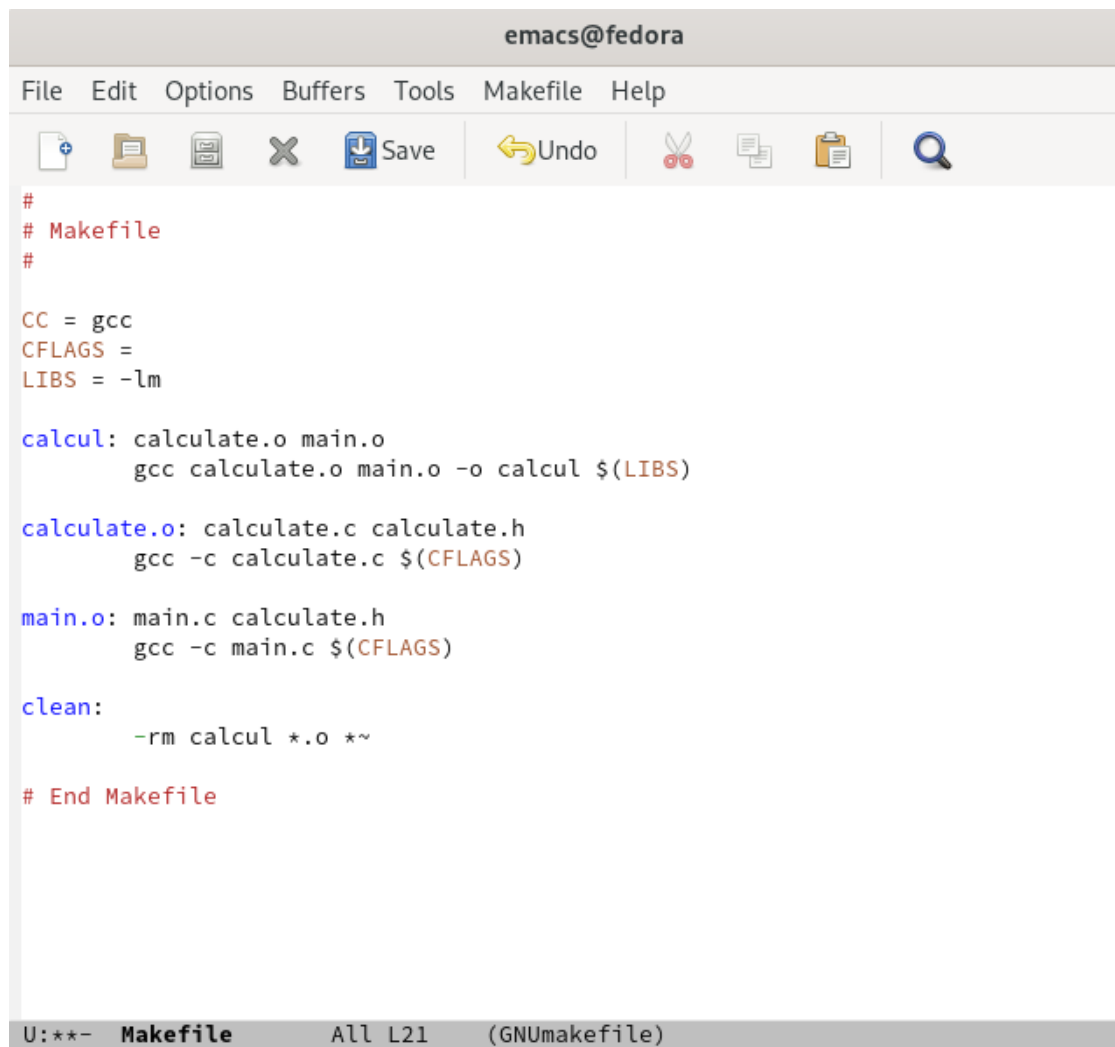


```
[mgandrianova@fedora lab_prog]$ gcc -c calculate.c  
[mgandrianova@fedora lab_prog]$ gcc -c main.c  
[mgandrianova@fedora lab_prog]$ gcc calculate.o main.o -o calcul -lm
```

Рис.7: Компиляция программы

4). В ходе компиляции программы никаких синтаксических ошибок выявлено не было.

5). Создала Makefile с необходимым содержанием(рис.8).



```
#
# 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 *~

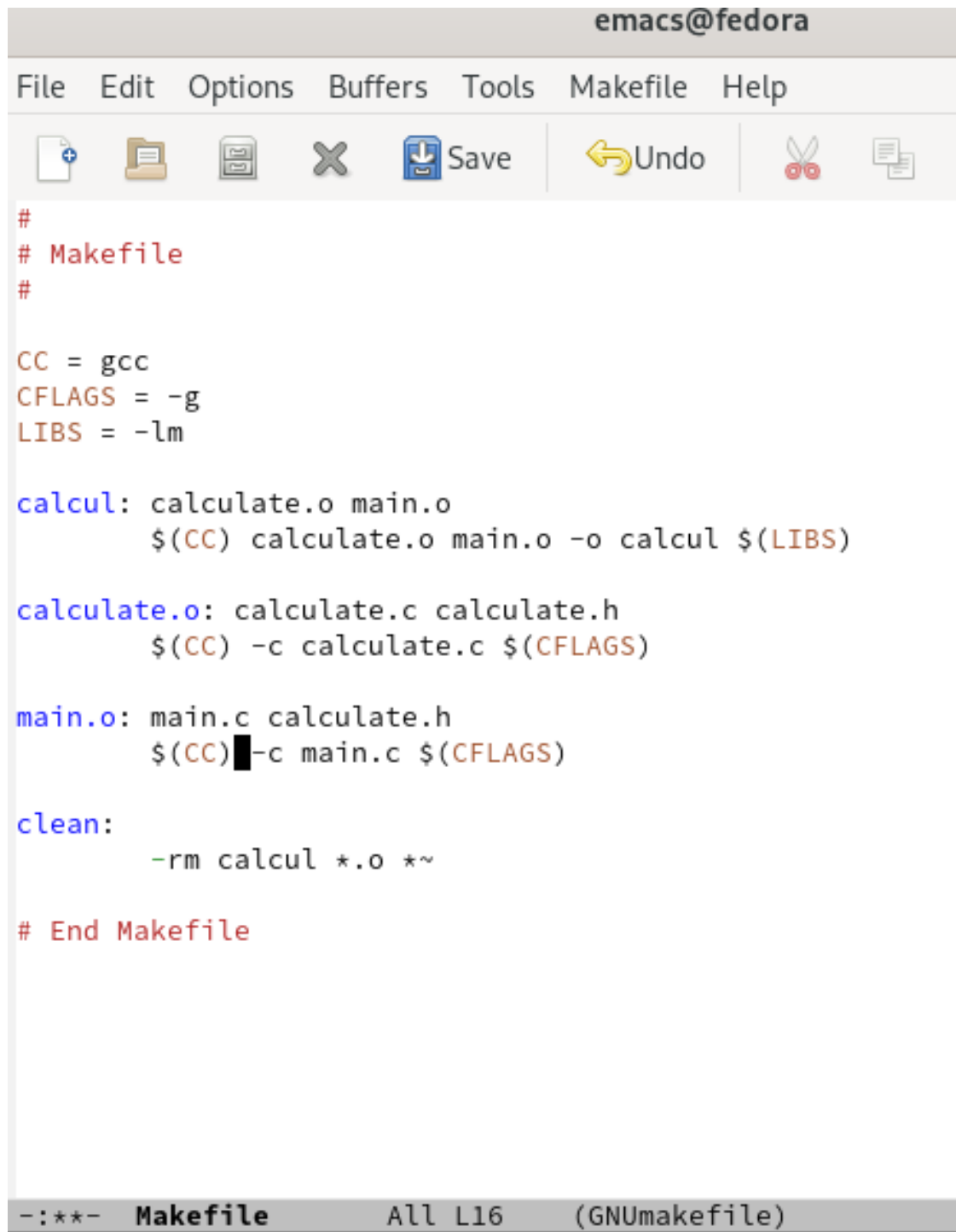
# End Makefile
```

U: ** - Makefile All L21 (GNUmakefile)

Рис.8: Программа в Makefile

Данный файл необходим для автоматической компиляции файлов calculate.c (цель calculate.o), main.c (цель main.o), а также их объединения в один исполняемый файл calcul (цель calcul). Цель clean нужна для автоматического удаления файлов. Переменная CC отвечает за утилиту для компиляции. Переменная CFLAGS отвечает за опции в данной утилите. Переменная LIBS отвечает за опции для объединения объектных файлов в один исполняемый файл.

6). Далее исправила Makefile (рис.9).



```
#
# Makefile
#

CC = gcc
CFLAGS = -g
LIBS = -lm

calcul: calculate.o main.o
    $(CC) calculate.o main.o -o calcul $(LIBS)

calculate.o: calculate.c calculate.h
    $(CC) -c calculate.c $(CFLAGS)

main.o: main.c calculate.h
    $(CC) -c main.c $(CFLAGS)

clean:
    -rm calcul *.o *~

# End Makefile
```

-:***- **Makefile** All L16 (GNUmakefile)

Рис.9: Изменённая программа в Makefile

В переменную CFLAGS добавила опцию -g, необходимую для компиляции объектных файлов и их использования в программе отладчика GDB. Сделала так, что утилита компиляции выбирается с помощью переменной CC. После этого я удалила исполняемые и объектные файлы из каталога с помощью команды “make clean”

(рис.10). Выполнила компиляцию файлов, используя команды “make calculate.o”, “make main.o”, “make calcul”(рис.11).

```
[mgandrianova@fedora lab_prog]$ make clean
rm calcul *.o *~
[2]+  Завершён          emacs
```

Рис.10: Удаление файлов

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

Рис.11: Компиляция файлов

С помощью gdb выполнила отладку программы calcul: – Запустила отладчик GDB, загрузив в него программу для отладки, используя команду: “gdb./calcul”(рис.12).

```
[mgandrianova@fedora lab_prog]$ gdb ./calcul
GNU gdb (GDB) Fedora 11.2-2.fc35
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...
(gdb) █
```

Рис.12: Работа с gdb

Для запуска программы внутри отладчика ввела команду “run”(рис.13).


```

(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/mgandrianova/work/os/lab_prog/calcul
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Число: 4
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): +
Второе слагаемое: 1
    5.00
[Inferior 1 (process 4247) exited normally]

```

Рис.13: Работа с *gdb* - *run*

Для постраничного (по 10 строк) просмотра исходного кода использовала команду "list" (рис.14).

```

(gdb) list
1      //////////////////////////////////////////
2      // main.c
3
4      #include <stdio.h>
5      #include "calculate.h"
6
7      int
8      main (void)
9      {
10         float Numeral;
(gdb) list
11         char Operation[4];
12         float Result;
13         printf("Число: ");
14         scanf("%f",&Numeral);
15         printf("Операция (+,-,*,/,pow,sqrt,sin,cos,tan): ");
16         scanf("%s",&Operation);
17         Result = calculate(Numeral, Operation);
18         printf("%6.2f\n",Result);
19         return 0;
20     }
(gdb) list
Line number 21 out of range; main.c has 20 lines.

```

Рис.14: Работа с *gdb* - *list*

Для просмотра строк с 12 по 15 основного файла использовала команду "list 12,15"(рис.15).

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

Рис.15: Работа с gdb - list 12,15

Для просмотра определённых строк не основного файла использовала команду "list calculate.c:20,29"(рис.16).

```
(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);
```

Рис.16: Работа с gdb - list calculate.c:20,29

Установила точку останова в файле calculate.c на строке номер 21, используя команды "list calculate.c:20,27" и "break 21"(рис.17).

```
(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.
```

Рис.17: Работа с gdb - list calculate.c:20,27 и break 21

Вывела информацию об имеющихся в проекте точках останова с помощью команды "info breakpoints"(рис.18).

```
(gdb) info breakpoints
Num      Type          Disp Enb Address                      What
1        breakpoint    keep y   0x00000000000040120f in Calculate at calculate.c:21
```

Рис.18: Работа с gdb - info breakpoints

Запустила программу внутри отладчика и убедилась, что программа остановилась в момент прохождения точки останова. Использовала команды "run", "5", "-" и "backtrace"(рис.19).Отладчик выдал следующую информацию(рис.20).

```
(gdb) run
Starting program: /home/mgandrianova/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=0x7fffffffdf14 "-") at calculate.c:21
21      printf("Вычитаемое: ");
(gdb) backtrace
```

Рис.19: Работа с gdb - run

```
#0 Calculate (Numeral=5, Operation=0x7fffffffdf14 "-") at calculate.c:21
#1 0x0000000000004014eb in main () at main.c:17
```

Рис.20: Результат после команды "backtrace"

Посмотрела, чему равно на этом этапе значение переменной Numeral, введя команду "print Numeral"(рис.21).

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

Рис.21: Работа с gdb - print Numeral

Сравнила с результатом вывода на экран после использования команды “display Numeral”(рис.22). Значения совпадают.

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

Рис.22: Работа с gdb - display Numeral

Убрала точки останова с помощью команд “info breakpoints” и “delete 1”(рис.23).

```
(gdb) info breakpoints
Num      Type           Disp Enb Address            What
1        breakpoint    keep y   0x00000000000040120f in calculate at calculate.c:21
          breakpoint already hit 1 time
(gdb) delete 1
```

Рис.23: Работа с gdb - info breakpoints

7). Далее воспользовалась командами “splint calculate.c” и “splint main.c” (рис.24,рис.25). С помощью утилиты splint выяснилось, что в файлах calculate.c и main.c присутствует функция чтения scanf, возвращающая целое число (тип int), но эти числа не используются и нигде не сохраняются. Утилита вывела предупреждение о том, что в файле calculate.c происходит сравнение вещественного числа с нулем. Также возвращаемые значения (тип double) в функциях pow, sqrt, sin, cos и tan записываются в переменную типа float, что свидетельствует о потере данных.

```

[mgandrianova@fedora lab_prog]$ splint calculate.c
Splint 3.1.2 --- 23 Jul 2021

calculate.h:7:37: 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:16:7: 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:22:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:28:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:34:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:35:10: 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:38:10: Return value type double does not match declared type float:
        (HUGE_VAL)
    To allow all numeric types to match, use +relaxtypes.
calculate.c:46:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:47:13: Return value type double does not match declared type float:
        (pow(Numeral, SecondNumeral))
calculate.c:50:11: Return value type double does not match declared type float:
        (sqrt(Numeral))
calculate.c:52:11: Return value type double does not match declared type float:
        (sin(Numeral))
calculate.c:54:11: Return value type double does not match declared type float:
        (cos(Numeral))
calculate.c:56:11: Return value type double does not match declared type float:
        (tan(Numeral))
calculate.c:60:13: Return value type double does not match declared type float:
        (HUGE_VAL)

Finished checking --- 15 code warnings

```

Рис.24: Результат команды splint calculate.c

```

[mgandrianova@fedora lab_prog]$ splint main.c
Splint 3.1.2 --- 23 Jul 2021

calculate.h:7:37: 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:14:3: 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:16:14: 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:16:11: Corresponding format code
main.c:16:3: Return value (type int) ignored: scanf("%s", &Ope...

Finished checking --- 4 code warnings

```

Рис.25: Результат команды splint main.c

Выводы

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.

Контрольные вопросы

- 1). Чтобы получить информацию о возможностях программ gcc, make, gdb и др. нужно воспользоваться командой man или опцией -help(-h) для каждой команды.
- 2). Процесс разработки программного обеспечения обычно разделяется на следующие этапы:
 1. планирование, включающее сбор и анализ требований к функционалу и другим характеристикам разрабатываемого приложения;
 2. проектирование, включающее в себя разработку базовых алгоритмов и спецификаций, определение языка программирования;
 3. непосредственная разработка приложения: окодирование – по сути создание исходного текста программы (возможно в нескольких вариантах); – анализ разработанного кода; сборка, компиляция и разработка исполняемого модуля; отестирование и отладка, сохранение произведённых изменений;
 4. 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.

3). Для имени входного файла суффикс определяет какая компиляция требуется. Суффиксы указывают на тип объекта. 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. Например, в команде «gcc -c main.c»: gcc по расширению (суффиксу).c распознает тип файла для компиляции и формирует объектный модуль – файл с расширением .o. Если требуется получить исполняемый файл с определённым именем (например, hello), то требуется воспользоваться опцией -o и в качестве параметра задать имя создаваемого файла: «gcc -o hello».

4). Основное назначение компилятора языка Си в UNIX заключается в компиляции всей программы и получении исполняемого файла/модуля.

5). Для сборки разрабатываемого приложения и собственно компиляции полезно воспользоваться утилитой make. Она позволяет автоматизировать процесс преобразования файлов программы из одной формы в другую, отслеживает взаимосвязи между файлами.

6). Для работы с утилитой make необходимо в корне рабочего каталога с Вашим проектом создать файл с названием makefile или Makefile, в котором будут описаны правила обработки файлов Вашего программного комплекса. В самом простом случае Makefile имеет следующий синтаксис: ... : ...<команда 1>... Сначала задаётся список целей, разделённых пробелами, за которым идёт двоеточие и список зависимостей. Затем в следующих строках указываются команды. Строки с командами обязательно должны начинаться с табуляции. В качестве цели в Makefile может выступать имя файла или название какого-то действия. Зависимость задаёт исходные параметры (условия) для достижения указанной цели. Зависимость также может быть названием какого-то действия. Команды – собственно действия, которые необходимо выполнить для достижения цели. Общий синтаксис Makefile имеет вид: target1 [target2...]:[:] [dependment1...][(tab) commands] [#commentary][(tab) commands] [#commentary]. Здесь знак # определяет начало комментария (содержимое от знака # и до конца строки не будет обрабатываться. Одинарное двоеточие указывает на то, что последовательность команд должна содержаться в одной строке. Для переноса можно в длинной строке команд можно использовать обратный слэш \. Двойное двоеточие указывает на то, что последовательность команд может содержаться в нескольких последовательных строках. Пример более сложного синтаксиса Makefile: ## Makefile for abcd.c #CC = gcc #CFLAGS = -g # Compile abcd.c normally abcd: abcd.c

(CFLAGS) abcd.c clean: -rm abcd.o ~ # End Makefile for abcd.c. В этом примере в начале файла заданы три переменные: CC и CFLAGS. Затем указаны цели, их зависимости и соответствующие команды. В командах происходит обращение к значениям переменных. Цель с именем clean производит очистку каталога от файлов,

полученных в результате компиляции. Для её описания использованы регулярные выражения.

7). Во время работы над кодом программы программист неизбежно сталкивается с появлением ошибок в ней. Использование отладчика для поиска и устранения ошибок в программе существенно облегчает жизнь программиста. В комплект программ GNU для ОС типа UNIX входит отладчик GDB (GNU Debugger). 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: gcc -g file.c. После этого для начала работы с gdb необходимо в командной строке ввести одноимённую команду, указав в качестве аргумента анализируемый бинарный файл: gdbfile.o

8). Основные команды отладчика gdb:

backtrace – вывод на экран пути к текущей точке останова (по сути вывод – названий всех функций);
break – установить точку останова (в качестве параметра может быть указан номер строки или название функции);
clear – удалить все точки останова в функции;
continue – продолжить выполнение программы;
delete – удалить точку останова;
display – добавить выражение в список выражений, значения которых отображаются при достижении точки останова программы;
finish – выполнить программу до момента выхода из функции;
info breakpoints – вывести на экран список используемых точек останова;
info watchpoints – вывести на экран список используемых контрольных выражений;
list – вывести на экран исходный код (в ходе выполнения данной лабораторной работы я приобрела простейшие навыки разработки, анализа, тестирования и отладки приложений в ОС типа UNIX/Linux на примере создания на языке программирования C калькулятора с простейшими функциями. в качестве параметра может быть указано название файла и через двоеточие номера начальной и конечной строк);
next – выполнить программу пошагово, но без выполнения вызываемых в программе функций;
print – вывести значение указываемого в качестве параметра выражения;
run – запуск программы на выполнение;
set – установить новое значение переменной;
step – пошаговое выполнение программы;
watch – установить контрольное выражение, при изменении значения которого программа будет остановлена. Для выхода из gdb можно воспользоваться командой quit (или её сокращённым вариантом q) или комбинацией клавиш Ctrl-d. Более подробную информацию по работе с gdb можно получить с помощью команд gdb-h и man gdb.

9). Схема отладки программы показана в 6 пункте лабораторной работы.

10). При первом запуске компилятор не выдал никаких ошибок, но в коде программы main.c допущена ошибка, которую компилятор мог пропустить

(возможно, из-за версии 8.3.0-19): в строке `scanf("%s", &Operation);` нужно убрать знак `&`, потому что имя массива символов уже является указателем на первый элемент этого массива.

11). 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). Утилита `splint` анализирует программный код, проверяет корректность задания аргументов использованных в программе функций и типов возвращаемых значений, обнаруживает синтаксические и семантические ошибки. В отличие от компилятора Си анализатор `splint` генерирует комментарии с описанием разбора кода программы и осуществляет общий контроль, обнаруживая такие ошибки, как одинаковые объекты, определённые в разных файлах, или объекты, чьи значения не используются в работе программы, переменные с некорректно заданными значениями и типами и многое другое.