

Лабораторная работа №13

НКАбд-01-22

Никита Михайлович Демидович

Содержание

1	Цель работы	5
2	Задание	6
3	Теоретическое введение	9
4	Выполнение лабораторной работы	11
5	Контрольные вопросы	23
6	Выводы	30
	Список литературы	31

Список иллюстраций

4.1	Создание подкаталога ~/work/os/lab_prog и файлов calculate.h, calculate.c и main.c	11
4.2	Исполняемый файл calculate.c	12
4.3	Исполняемый файл calculate.h	15
4.4	Исполняемый файл main.h	16
4.5	Процесс компиляции программы	17
4.6	Текст Makefile	18
4.7	Отладка программы с помощью gdb	19
4.8	Работа программы	19
4.9	Построчный просмотр кода программы	20
4.10	Построчный просмотр кода программы	20
4.11	Отладка программы с помощью gdb	20
4.12	Отладка программы с помощью gdb	21
4.13	Отладка программы с помощью gdb	21
4.14	Код программы calculate.c в утилите splint	22
4.15	Код программы main.c в утилите splint	22

Список таблиц

1 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.

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`.
6. С помощью `gdb` выполните отладку программы `calcul` (перед использованием `gdb` исправьте `Makefile`):
 - Запустите отладчик GDB, загрузив в него программу для отладки:

```
gdb ./calcul
```

- Для запуска программы внутри отладчика введите команду `run`.
- Для постраничного (по 9 строк) просмотра исходного код используйте команду:

```
1 list
```

- Для просмотра строк с 12 по 15 основного файла используйте `list` с параметрами:

```
list 12,15
```

- Для просмотра определённых строк не основного файла используйте `list` с параметрами:

```
list calculate.c:20,29
```

- Установите точку останова в файле `calculate.c` на строке номер 21:

```
list calculate.c:20,27 break 21
```

- Выведите информацию об имеющихся в проекте точках останова:

```
info breakpoints
```

- Запустите программу внутри отладчика и убедитесь, что программа останавливается в момент прохождения точки останова.
- Отладчик выдаст следующую информацию:

```
#0 Calculate (Numeral=5, Operation=0x7fffffff280 "-")    at calculate.c:21
#1 0x0000000000400b2b in main () at main.c:17
```

а команда `backtrace` покажет весь стек вызываемых функций от начала программы до текущего места.

- Посмотрите, чему равно на этом этапе значение переменной `Numeral`, введя:

```
print Numeral
```

На экран должно быть выведено число 5.

- Сравните с результатом вывода на экран после использования команды:

`display Numeral`

- Уберите точки останова.

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:

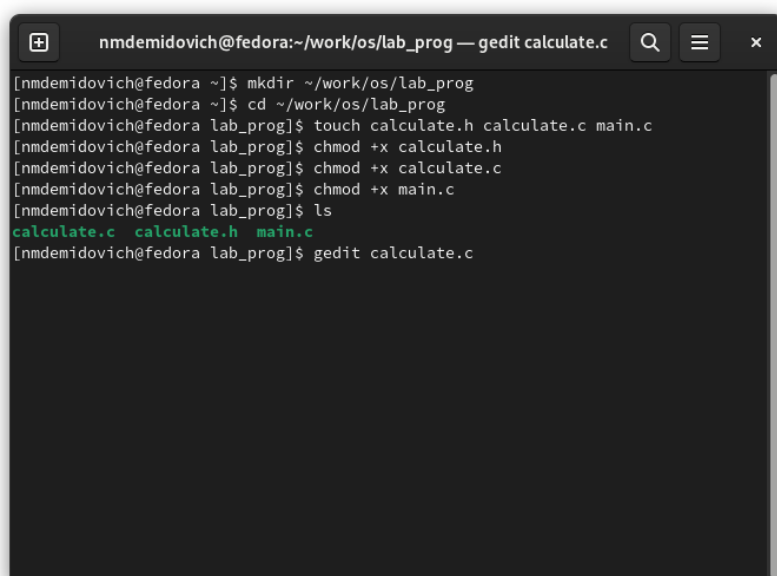
- планирование, включающее сбор и анализ требований к функционалу и другим характеристикам разрабатываемого приложения;
- проектирование, включающее в себя разработку базовых алгоритмов и спецификаций, определение языка программирования;
- непосредственная разработка приложения;
- кодирование — по сути 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 gсс, которая интерпретирует аргументы командной строки, определяет и осуществляет запуск нужного компилятора для входного файла. Файлы с расширением (суффиксом) .с воспринимаются gсс как программы на языке C, файлы с расширением .сс или .C — как файлы на языке C++, а файлы с расширением .о считаются объектными.

4 Выполнение лабораторной работы

На первом этапе выполнения работы я создал подкаталог `~/work/os/lab_prog` и все необходимые файлы, после чего написал программы и сделал эти файлы исполняемыми (рис. [4.1]) - (рис. [4.4]):

A screenshot of a terminal window with a dark background. The title bar at the top reads "nmdemidovich@fedora:~/work/os/lab_prog — gedit calculate.c". The terminal shows a series of commands and their outputs: creating a directory, changing to it, creating three files, setting permissions, listing files, and opening one of them in gedit.

```
nmdemidovich@fedora ~]$ mkdir ~/work/os/lab_prog
nmdemidovich@fedora ~]$ cd ~/work/os/lab_prog
nmdemidovich@fedora lab_prog]$ touch calculate.h calculate.c main.c
nmdemidovich@fedora lab_prog]$ chmod +x calculate.h
nmdemidovich@fedora lab_prog]$ chmod +x calculate.c
nmdemidovich@fedora lab_prog]$ chmod +x main.c
nmdemidovich@fedora lab_prog]$ ls
calculate.c calculate.h main.c
nmdemidovich@fedora lab_prog]$ gedit calculate.c
```

Рис. 4.1: Создание подкаталога `~/work/os/lab_prog` и файлов `calculate.h`, `calculate.c` и `main.c`

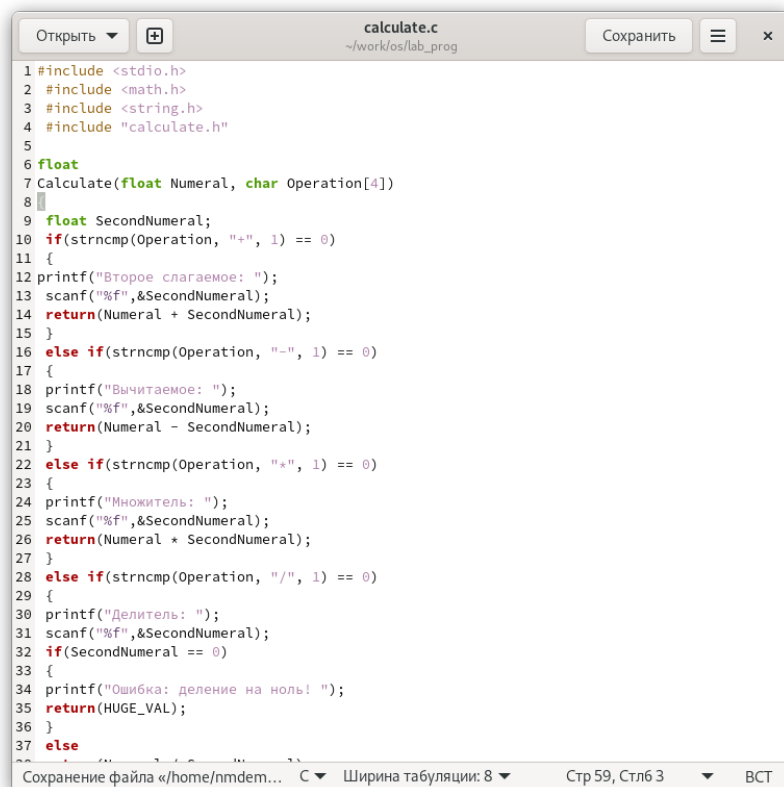


Рис. 4.2: Исполняемый файл calculate.c

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

float
Calculate(float Numeral, char Operation[4])
{
    float SecondNumeral;
    if(strcmp(Operation, "+", 1) == 0)
    {
        printf("Второе слагаемое: ");
        scanf("%f", &SecondNumeral);
        return(Numeral + SecondNumeral);
    }
    else if(strcmp(Operation, "-", 1) == 0)
    {
        printf("Вычитаемое: ");
        scanf("%f", &SecondNumeral);
        return(Numeral - SecondNumeral);
    }
    else if(strcmp(Operation, "*", 1) == 0)
    {
        printf("Множитель: ");
        scanf("%f", &SecondNumeral);
        return(Numeral * SecondNumeral);
    }
    else if(strcmp(Operation, "/", 1) == 0)
    {
        printf("Делитель: ");
        scanf("%f", &SecondNumeral);
        if(SecondNumeral == 0)
        {
            printf("Ошибка: деление на ноль! ");
            return(HUGE_VAL);
        }
    }
    else
```

```

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);
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);
}
}

```

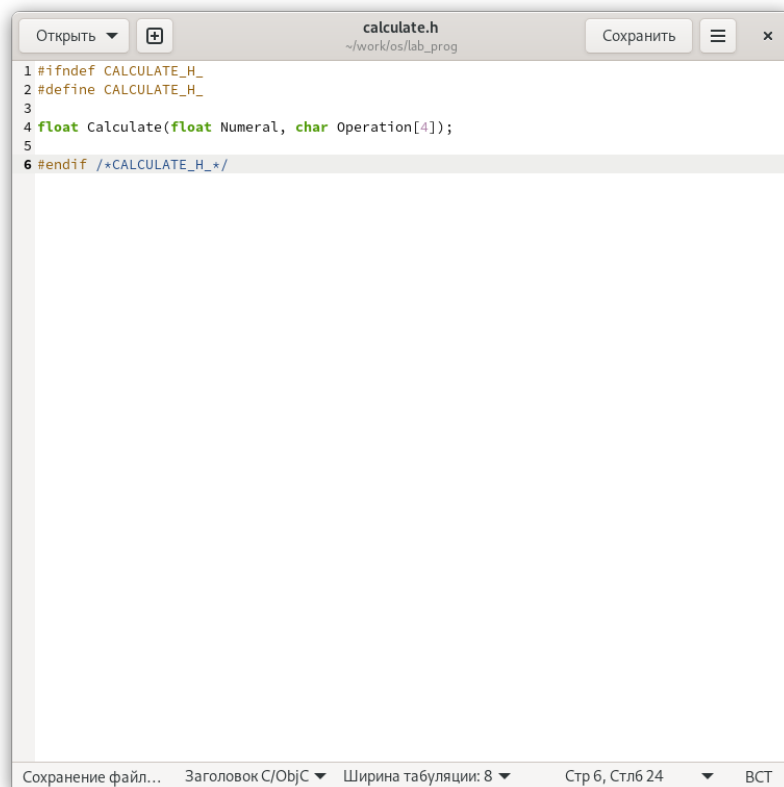


Рис. 4.3: Исполняемый файл calculate.h

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

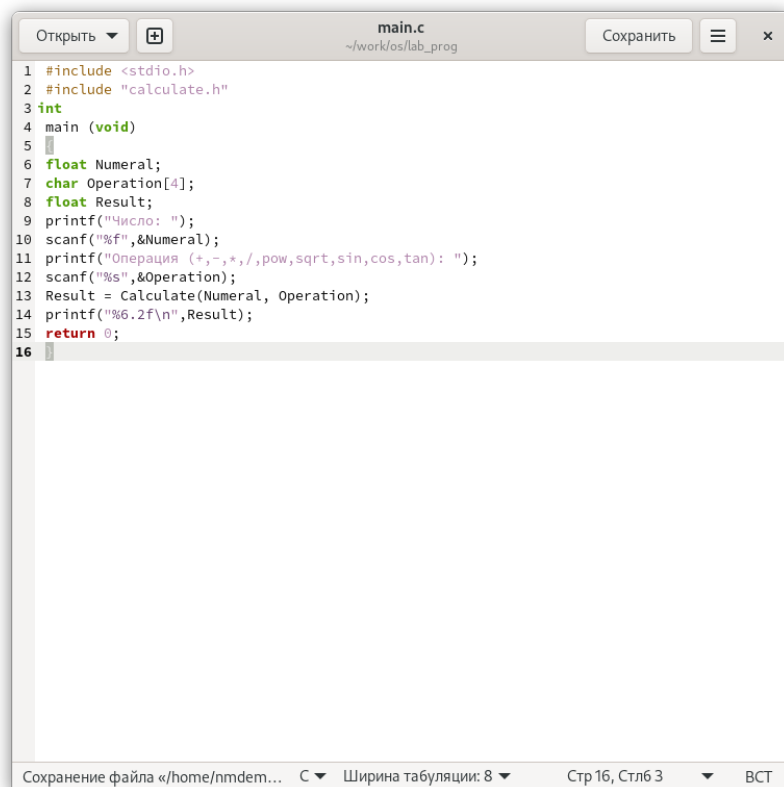


Рис. 4.4: Исполняемый файл main.h

```
#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);
```

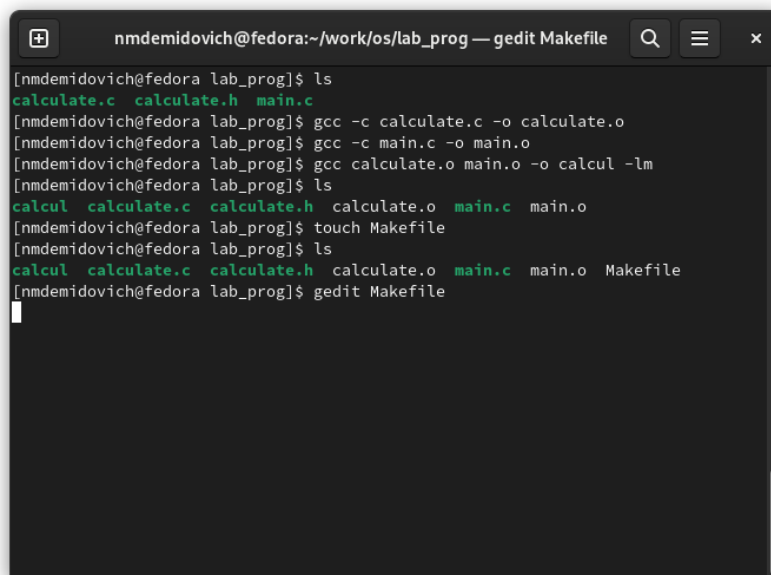


```

Result = Calculate(Numeral, Operation);
printf("%6.2f\n", Result);
return 0;
}

```

Затем я выполнил компиляцию программы посредством gcc и создал Makefile (рис. [4.5]) - (рис. [4.6]):

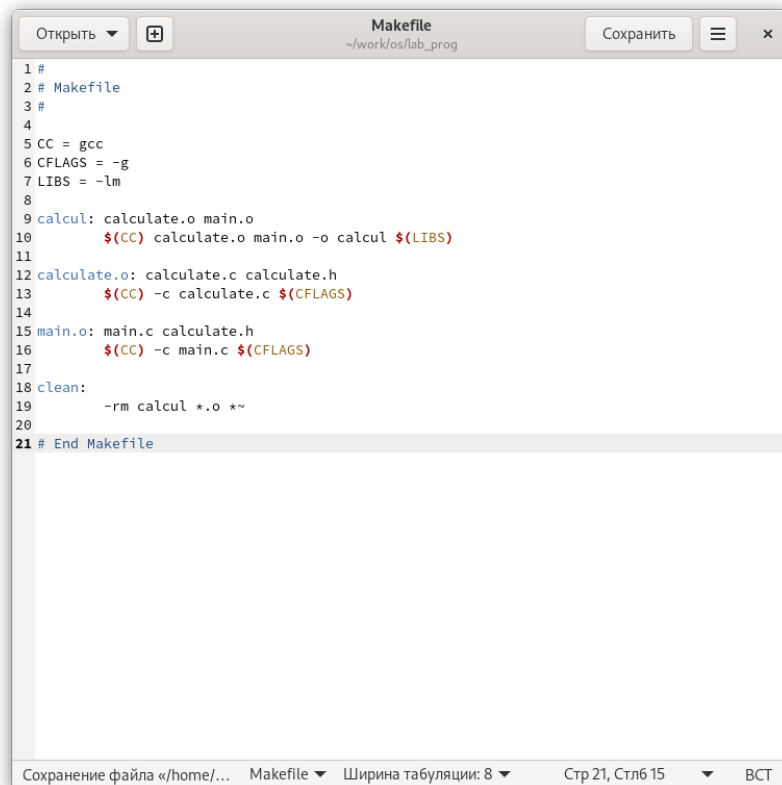


```

nmdemidovich@fedora:~/work/os/lab_prog — gedit Makefile
[nmdemidovich@fedora lab_prog]$ ls
calculate.c calculate.h main.c
[nmdemidovich@fedora lab_prog]$ gcc -c calculate.c -o calculate.o
[nmdemidovich@fedora lab_prog]$ gcc -c main.c -o main.o
[nmdemidovich@fedora lab_prog]$ gcc calculate.o main.o -o calcul -lm
[nmdemidovich@fedora lab_prog]$ ls
calcul calculate.c calculate.h calculate.o main.c main.o
[nmdemidovich@fedora lab_prog]$ touch Makefile
[nmdemidovich@fedora lab_prog]$ ls
calcul calculate.c calculate.h calculate.o main.c main.o Makefile
[nmdemidovich@fedora lab_prog]$ gedit Makefile

```

Рис. 4.5: Процесс компиляции программы



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

Рис. 4.6: Текст Makefile

После этого я с помощью gdb выполнил отладку программы calcul (перед использованием gdb исправил Makefile) (рис. [4.7]):

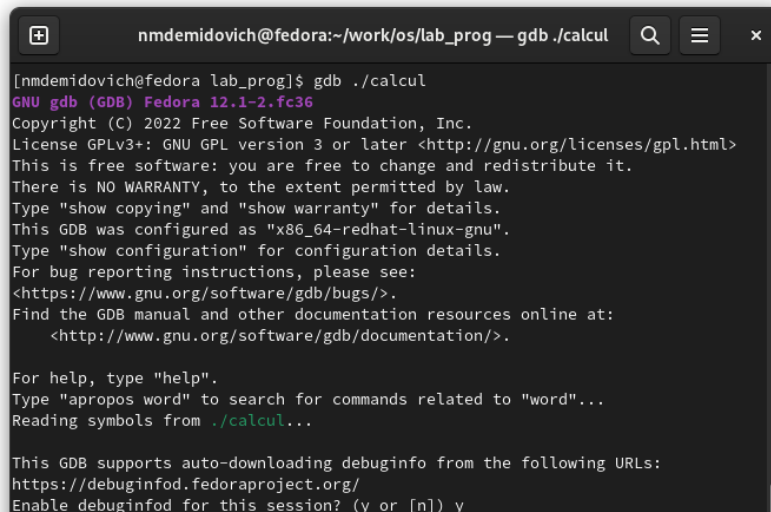
A terminal window titled 'nmdemidovich@fedora:~/work/os/lab_prog — gdb ./calcul'. The prompt is '[nmdemidovich@fedora lab_prog]\$'. The user enters 'gdb ./calcul'. The terminal displays the GNU GDB (GDB) Fedora 12.1-2.fc36 startup screen, including copyright information, license details (GPLv3+), and instructions for using GDB. It also shows the GDB configuration as 'x86_64-redhat-linux-gnu' and provides links for bug reporting and documentation. The prompt returns to '[nmdemidovich@fedora lab_prog]\$'.

Рис. 4.7: Отладка программы с помощью gdb

Далее я проверил её работу (рис. [4.8]):

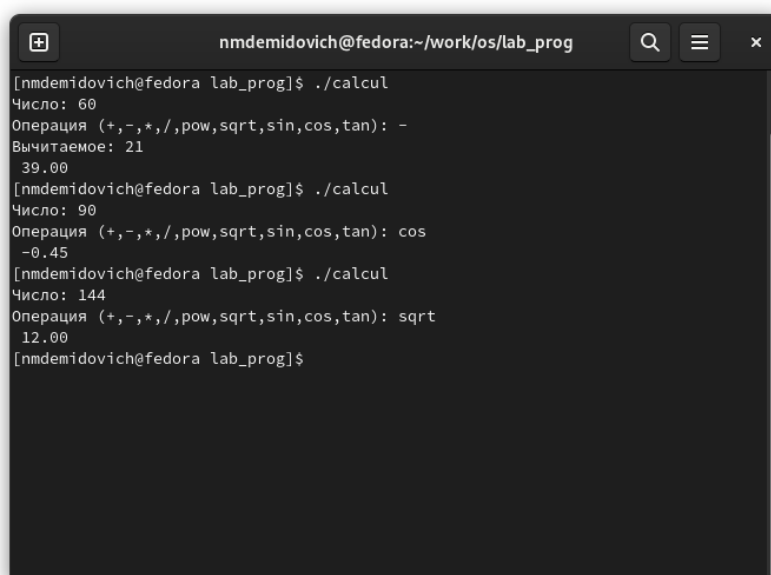
A terminal window titled 'nmdemidovich@fedora:~/work/os/lab_prog'. The prompt is '[nmdemidovich@fedora lab_prog]\$'. The user enters './calcul'. The program outputs 'Число: 60' and 'Операция (+,-,*,/,pow,sqrt,sin,cos,tan): -'. The user enters '21'. The program outputs '39.00'. The user enters './calcul'. The program outputs 'Число: 90' and 'Операция (+,-,*,/,pow,sqrt,sin,cos,tan): cos'. The user enters 'cos'. The program outputs '-0.45'. The user enters './calcul'. The program outputs 'Число: 144' and 'Операция (+,-,*,/,pow,sqrt,sin,cos,tan): sqrt'. The user enters 'sqrt'. The program outputs '12.00'. The prompt returns to '[nmdemidovich@fedora lab_prog]\$'.

Рис. 4.8: Работа программы

После чего я построчно (по 9 строк) просмотрел код программы (рис. [4.9]):

```
(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. 4.9: Построчный просмотр кода программы

Затем я просмотрел определённые строки не основного файла (рис. [4.10]):

```
(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. 4.10: Построчный просмотр кода программы

Далее я выполнил следующие задания:

- Установил точку останова в файле calculate.c на строке номер 21:

```
list calculate.c:20,27 break 21
```

- Вывел информацию об имеющихся в проекте точках останова:

```
info breakpoints
```

- Запустил программу внутри отладчика и убедился, что программа остановится в момент прохождения точки останова.

```
#0 Calculate (Numeral=5, Operation=0x7fffffff934 "-") at calculate.c:21
```

```
#1 0x0000000000400b2b in main () at main.c:16
```

(рис. [4.11]):

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

Рис. 4.11: Отладка программы с помощью gdb

- Посмотрел, чему равно на этом этапе значение переменной Numeral, введя:

```
print Numeral
```

На экран, как и должно было быть, вывелось число 5 (рис. [4.12]):

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

Рис. 4.12: Отладка программы с помощью gdb

- Сравнил с результатом вывода на экран после использования команды:

```
display Numeral
```

- Убрал точки останова (рис. [4.13]):

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

Рис. 4.13: Отладка программы с помощью gdb

- С помощью утилиты splint попробуйте проанализировать коды файлов calculate.c и main.c (рис. [4.14]) - (рис. [4.15]):

```

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. 4.14: Код программы calculate.c в утилите splint

```

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. 4.15: Код программы main.c в утилите splint

5 Контрольные вопросы

1. Как получить информацию о возможностях программ gcc, make, gdb и др.?

Дополнительную информацию о 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 “.с” 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 .с компилятор распознает, что файл abcd.c должен компилироваться, а по суффиксу .о, 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 С 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:

```
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, начина 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. Второй способ 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 передается 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.

1. Выполнили компиляцию программы 2) Увидели ошибки в программе
2. Открыли редактор и исправили программу
3. Загрузили программу в отладчик `gdb run` — отладчик выполнил программу, мы ввели требуемые значения.
4. 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.

6 Выводы

В результате выполнения данной лабораторной работы я приобрел простейшие навыки разработки, анализа, тестирования и отладки приложений в ОС типа UNIX/Linux на примере создания на языке программирования С калькулятора с простейшими функциями.

Список литературы

Лабораторная работа №13 (Архитектура ОС)