

**Лабораторная работа №13. Средства,
применяемые при разработке
программного обеспечения в ОС типа
UNIX/Linux.**

Операционные системы

Кочарян Никита Робертович

Содержание

1	Цель работы	5
2	Задания	6
3	Выполнение лабораторной работы	8
4	Контрольные вопросы	13
5	Ответы на контрольные вопросы	14
6	Вывод	18

Список иллюстраций

3.1	рис1	8
3.2	рис2	8
3.3	рис3	9
3.4	рис4	9
3.5	рис5	9
3.6	рис6	10
3.7	рис7	10
3.8	рис8	10
3.9	рис9	11
3.10	рис10	11
3.11	рис11	12

Список таблиц

1 Цель работы

Приобрести простейшие навыки разработки, анализа, тестирования и отладки приложений в ОС типа UNIXLinux на примере создания на языке программирования С калькулятора с простейшими функциями.

2 Задания

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: Поясните в отчёте его содержание
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: run` – Для постраничного (по 9 строк) просмотра исходного кода используйте команду `list: 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 точке останова: `info breakpoints` – Запустите программу внутри отладчика и убедитесь, что программа остановится в момент прохождения точки останова: `run`

- `backtrace` Отладчик выдаст следующую информацию: `#0 Calculate (Numeral=5, Operation=0x7fffffff280 "-") at calculate.c:21 #1 0x000000000400b2b 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 Numera` – Уберите точки останова: `info breakpoints delete 1`
7. С помощью утилиты `splint` попробуйте проанализировать коды файлов `calculate.c` и `main.c`.

3 Выполнение лабораторной работы

1. В домашнем каталог создал подкаталог ~/work/os/lab_prog.

```
nrkocharyan@dk4n62 ~ $ mkdir work/os/  
nrkocharyan@dk4n62 ~ $ cd work/os  
nrkocharyan@dk4n62 ~/work/os $ mkdir lab_prog  
nrkocharyan@dk4n62 ~/work/os $ cd lab_prog
```

Рис. 3.1: рис1

2. Создаю в нем 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.

```
nrkocharyan@dk4n62 ~/work/os/lab_prog $ touch calculate.h  
nrkocharyan@dk4n62 ~/work/os/lab_prog $ touch calculate.c  
nrkocharyan@dk4n62 ~/work/os/lab_prog $ touch main.c  
nrkocharyan@dk4n62 ~/work/os/lab_prog $ ls  
calculate.c calculate.h main.c  
nrkocharyan@dk4n62 ~/work/os/lab_prog $
```

Рис. 3.2: рис2

3. Реализация функций калькулятора в файле calculate.h.


```

1 #include <stdio.h>
2 #include <math.h>
3 #include <string.h>
4 #include "calculate.h"
5
6 float
7 calculate(float Numeral, char Operation[4])
8 {
9     float SecondNumeral;
10    if(strncmp(Operation, "+", 1) == 0)
11    {
12        printf("Сложение: ");
13        scanf("%f", &SecondNumeral);
14        return(Numeral + SecondNumeral);
15    }
16    else if(strncmp(Operation, "-", 1) == 0)
17    {
18        printf("Вычитание: ");
19        scanf("%f", &SecondNumeral);
20        return(Numeral - SecondNumeral);
21    }
22    else if(strncmp(Operation, "*", 1) == 0)
23    {
24        printf("Умножение: ");
25        scanf("%f", &SecondNumeral);
26        return(Numeral * SecondNumeral);
27    }
28    else if(strncmp(Operation, "/", 1) == 0)
29    {
30        printf("Деление: ");
31        scanf("%f", &SecondNumeral);
32        if(SecondNumeral == 0)
33        {
34            printf("Ошибка: деление на ноль! ");
35            return(HUGE_VAL);
36        }
37        else
38            return(Numeral / SecondNumeral);
39    }
40    else if(strncmp(Operation, "^", 1) == 0)
41    {
42        printf("Степень: ");
43        scanf("%f", &SecondNumeral);
44        return(pow(Numeral, SecondNumeral));
45    }
46    else if(strncmp(Operation, "sqrt", 4) == 0)
47        return(sqrt(Numeral));
48    }

```

Рис. 3.3: рис3

```

45    }
46    else if(strncmp(Operation, "sqrt", 4) == 0)
47        return(sqrt(Numeral));
48    else if(strncmp(Operation, "sin", 3) == 0)
49        return(sin(Numeral));
50    else if(strncmp(Operation, "cos", 3) == 0)
51        return(cos(Numeral));
52    else if(strncmp(Operation, "tan", 3) == 0)
53        return(tan(Numeral));
54    else
55    {
56        printf("Неправильно введено действие ");
57        return(HUGE_VAL);
58    }
59 }

```

Рис. 3.4: рис4

4. Реализация интерфейсного файла calculate.h, описывающий формат вызова функции-калькулятор

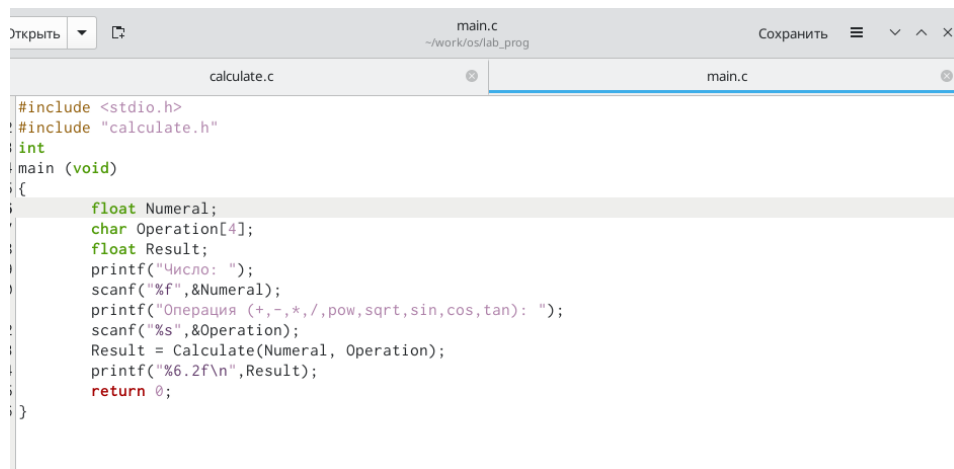
```

1 #ifndef CALCULATE_H_
2 #define CALCULATE_H_
3
4 float Calculate(float Numeral, char Operation[4]);
5
6 #endif /*CALCULATE_H_*/

```

Рис. 3.5: рис5

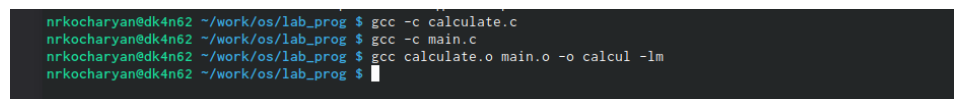
5. Реализация основного файла main.c, реализующая интерфейс пользователя к калькулятору



```
#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;
}
```

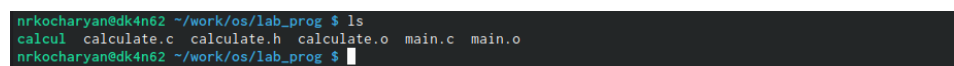
Рис. 3.6: рис6

6. Выполняю компиляцию программы посредством gcc



```
nrkocharyan@dk4n62 ~/work/os/lab_prog $ gcc -c calculate.c
nrkocharyan@dk4n62 ~/work/os/lab_prog $ gcc -c main.c
nrkocharyan@dk4n62 ~/work/os/lab_prog $ gcc calculate.o main.o -o calcul -lm
nrkocharyan@dk4n62 ~/work/os/lab_prog $
```

Рис. 3.7: рис7

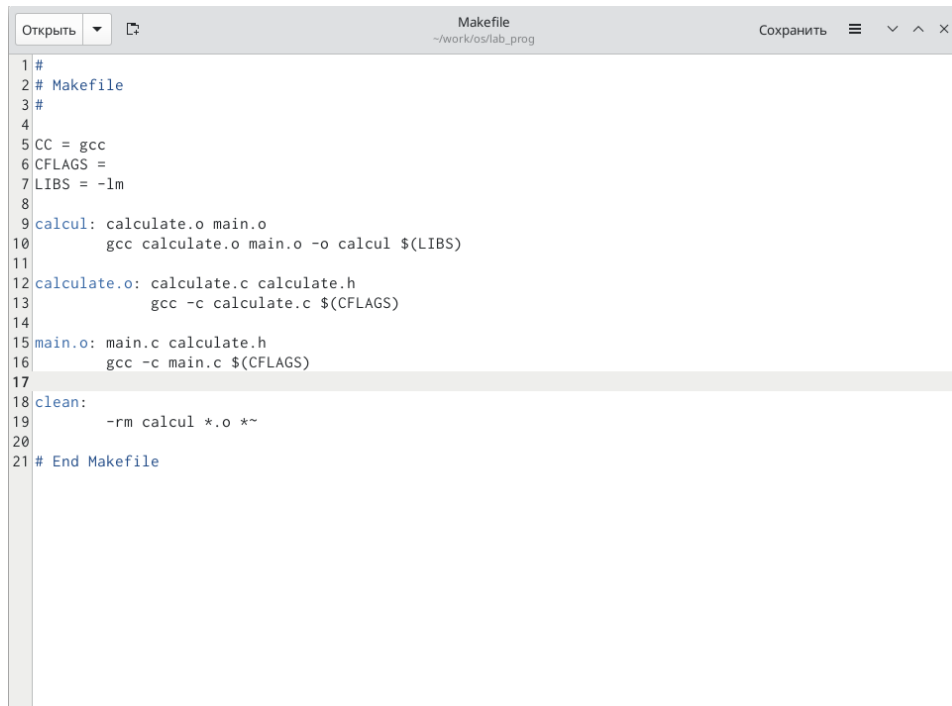


```
nrkocharyan@dk4n62 ~/work/os/lab_prog $ ls
calcul calculate.c calculate.h calculate.o main.c main.o
nrkocharyan@dk4n62 ~/work/os/lab_prog $
```

Рис. 3.8: рис8

7. Исправил синтаксические ошибки

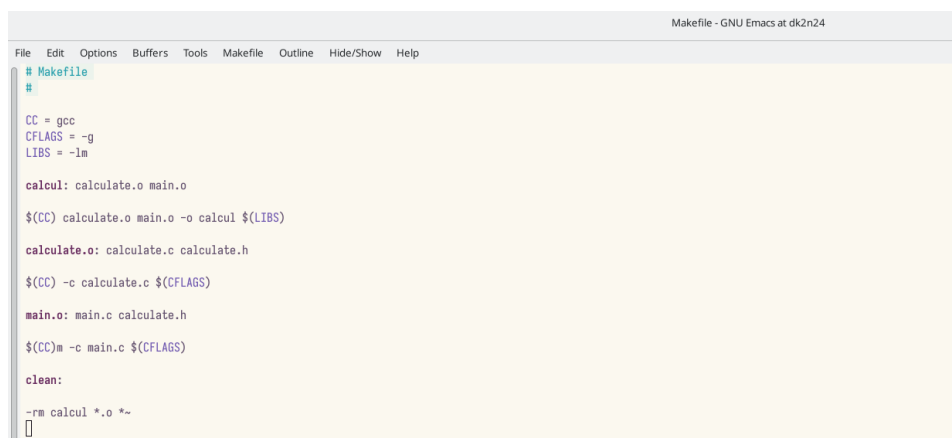
8. Создаю Makefile



```
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
```

Рис. 3.9: рис9

9. С помощью gdb выполняю отладку программы calcul (перед использованием gdb исправляю Makefile): Запускаю отладчик GDB, загрузив в него программу для отладки gdb ./calcul



```
# 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 *~
```

Рис. 3.10: рис10

```

nrkocharyan@dk4n62 ~/work/os/lab_prog $ gdb ./calcul
GNU gdb (Gentoo 12.1 vanilla) 12.1
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-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://bugs.gentoo.org/>.
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...
(No debugging symbols found in ./calcul)
(gdb) run
Starting program: /afs/.dk.sci.pfu.edu.ru/home/n/r/nrkocharyan/work/os/lab_prog/calcul
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib64/libthread_db.so.1".
Число: 3
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): -
Вычитаемое: 2
1.00
[Inferior 1 (process 7658) exited normally]
(gdb)

```

Рис. 3.11: рис11

4 Контрольные вопросы

1. Как получить информацию о возможностях программ gcc, make, gdb и др.?
2. Назовите и дайте краткую характеристику основным этапам разработки приложений в UNIX.
3. Что такое суффикс в контексте языка программирования? Приведите примеры ис- пользования.
4. Каково основное назначение компилятора языка C в UNIX?
5. Для чего предназначена утилита make?
6. Приведите пример структуры Makefile. Дайте характеристику основным элементам этого файла.
7. Назовите основное свойство, присущее всем программам отладки. Что необходимо сделать, чтобы его можно было использовать?
8. Назовите и дайте основную характеристику основным командам отладчика gdb.
9. Опишите по шагам схему отладки программы, которую Вы использовали при выполнении лабораторной работы.
10. Прокомментируйте реакцию компилятора на синтаксические ошибки в программе при его первом запуске.
11. Назовите основные средства, повышающие понимание исходного кода программы.
12. Каковы основные задачи, решаемые программой splint?

5 Ответы на контрольные вопросы

1. Информацию об этих программах можно получить с помощью функций `info` и `man`.
2. 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. 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. Основное назначение компилятора с языка Си заключается в компиляции всей программы в целом и получении исполняемого модуля.
5. 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. 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. Второй способ позволяет включать в исполняемый модуль testabcd возможность выполнить процесс отладки на уровне исходного текста. Пример можно найти в задании 5.

7. 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. 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 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

9. Выполнил компиляцию программы 2) Увидела ошибки в программе 3) Открыл редактор
10. Отладчику не понравился формат `%s` для `&Operation`, т.к `%s` — символьный формат, а значит необходим только `Operation`.
11. 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: – `cscore` - исследование функций, содержащихся в программе; – `splint` — критическая проверка программ, написанных на языке Си.
12. Проверка корректности задания аргументов всех использованных в программе

6 Вывод

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