

Отчёт по лабораторной работе №13

дисциплина: Операционные системы

Максим Александрович Мишонков

Содержание

1	Цель работы	4
2	Задание	5
3	Теоретическое введение	6
4	Выполнение лабораторной работы	7
5	Выводы	22

Список иллюстраций

4.1	Создание каталога и файлов	7
4.2	Скрипт	8
4.3	Скрипт	9
4.4	Скрипт	10
4.5	Скрипт	10
4.6	Компиляция программы	11
4.7	Скрипт	11
4.8	Скрипт	12
4.9	Команда “make clean”	12
4.10	Компиляция файлов	13
4.11	Отладчик	13
4.12	Команда “list”	14
4.13	Различные команды	15
4.14	Запуск программы	16
4.15	Анализ файла	16
4.16	Анализ файла	17
4.17	Аналих файла	17

1 Цель работы

Целью данной лабораторной работы является приобретение простейших навыков разработки, анализа, тестирования и отладки приложений в ОС типа UNIX/Linux на примере создания на языке программирования С калькулятора с простейшими функциями.

2 Задание

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.

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.

Отладка — этап разработки компьютерной программы, на котором обнаруживают, локализуют и устраняют ошибки. Чтобы понять, где возникла ошибка, приходится узнавать текущие значения переменных и выяснять, по какому пути выполнялась программа.

4 Выполнение лабораторной работы

1. Создал рабочее пространство для данной лабораторной работы и необходимые файлы. (рис. [4.1])

```
mamishonkov@dk8n53 ~ $ mkdir -p ~/work/os/lab_prog
mamishonkov@dk8n53 ~ $ cd ~/work/os/lab_prog
mamishonkov@dk8n53 ~/work/os/lab_prog $ touch calculate.h calculate.c main.c
mamishonkov@dk8n53 ~/work/os/lab_prog $ ls
calculate.c calculate.h main.c
mamishonkov@dk8n53 ~/work/os/lab_prog $
```

Рис. 4.1: Создание каталога и файлов

2. Написал текст программы в файле calculate.c. (рис. [4.2],[4.3])

```

////////////////////////////////////
// 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);
        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);
        }
    }
}

```

Рис. 4.2: Скрипт


```

}
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: Скрипт

3. Написал текст программы в интерфейсном файле calculate.h, описывающем формат вызова функции калькулятора. (рис. [4.4])

```

////////////////////////////////////
// calculate.h

#ifndef CALCULATE_H_
#define CALCULATE_H_

float Calculate(float Numeral, char Operation[4]);

#endif /*CALCULATE_H_*/

```

Рис. 4.4: Скрипт

4. Написал текст программы в основном файле main.c, реализующем интерфейс пользователя к калькулятору. (рис. [4.5])

```

////////////////////////////////////
// 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. 4.5: Скрипт

5. Выполнил компиляцию программы посредством gcc. (рис. [4.6])

```

mamishonkov@dk8n53 ~/work/os/lab_prog $ gcc -c calculate.c
[3]+  Завершён      emacs
mamishonkov@dk8n53 ~/work/os/lab_prog $ gcc -c main.c
mamishonkov@dk8n53 ~/work/os/lab_prog $ gcc calculate.o main.o -o calcul -lm

```

Рис. 4.6: Компиляция программы

6. Создал makefile с необходимым содержанием, который необходим для автоматической компиляции файлов calculate.c, main.c, а также их объединения в один исполняемый файл calcul. (рис. [4.7])

```

#
# 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

```

Рис. 4.7: Скрипт

7. Изменил файл, добавив в переменную CEFAGS “-g”, которая необходима для компиляции объектных файлов и их использования в программе отладчика GDB. (рис. [4.8])

```

#
# 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

```

Рис. 4.8: Скрипт

8. Удалил исполняемый и объектный файлы из каталога при помощи команды “make clean”. (рис. [4.9])

```

mamishonkov@dk8n53 ~/work/os/lab_prog $ make clean
rm calcul *.o *~
[3]+  Завершён          emacs
mamishonkov@dk8n53 ~/work/os/lab_prog $

```

Рис. 4.9: Команда “make clean”

9. Выполнил компиляцию файлов. (рис. [4.10])

```

mamishonkov@dk8n53 ~/work/os/lab_prog $ make calculate.o
gcc -c calculate.c -g
mamishonkov@dk8n53 ~/work/os/lab_prog $ make main.o
gcc -c main.c -g
mamishonkov@dk8n53 ~/work/os/lab_prog $ make calcul
gcc calculate.o main.o -o calcul -lm
mamishonkov@dk8n53 ~/work/os/lab_prog $ 

```

Рис. 4.10: Компиляция файлов

10. Выполнил gdb отладку программы calcul. Ввёл команду “run” для запуска программы внутри отладчика. (рис. [4.11])

```

mamishonkov@dk8n53 ~/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...
(gdb) run
Starting program: /afs/.dk.sci.pfu.edu.ru/home/m/a/mamishonkov/work/os/lab_prog/calcul
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib64/libthread_db.so.1".
Число: 6
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): *
Множитель: 5
30.00
[Inferior 1 (process 9481) exited normally]
(gdb) 

```

Рис. 4.11: Отладчик

11. Для построичного просмотра исходного кода использовал команду “list”. (рис. [4.12])

```

(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) 

```

Рис. 4.12: Команда “list”

12. Проверил команды list (для просмотра строк основного файла), list calculate.c (для просмотра определённых строк не основного файла), break (для установки точки в файле), info breakpoints (для вывода информации об имеющихся точках останова). (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) 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) info breakpoints
No breakpoints or watchpoints.
(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 0x555555555247: file calculate.c, line 21.
(gdb) info breakpoints
Num   Type             Disp Enb Address            What
1      breakpoint      keep y   0x0000555555555247 in Calculate at calculate.c:21

```

Рис. 4.13: Различные команды

13. Запустил программу внутри отладчика и убедился, что программа остановилась в момент прохождения точки останова. Посмотрел, чем равно на этом этапе значение переменной Numeral и сравнил его с результатом вывода на экран. Значения совпадают. Затем убрал точки останова. (рис. [4.14])

```

(gdb) run
Starting program: /afs/.dk.sci.pfu.edu.ru/home/m/a/mamishonkov/work/os/lab_prog/calcul
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib64/libthread_db.so.1".
Число: 5
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): -

Breakpoint 1, Calculate (Numeral=5, Operation=0x7fffffffdb4 "-") at calculate.c:21
21      printf("Вычитаемое: ");
(gdb) backtrace
#0 Calculate (Numeral=5, Operation=0x7fffffffdb4 "-") at calculate.c:21
#1 0x00005555555555a5 in main () at main.c:17
(gdb) print Numeral
$1 = 5
(gdb) display Numeral
1: Numeral = 5
(gdb) info breakpoints
Num      Type             Disp Enb Address                  What
1        breakpoint      keep y   0x00005555555555247 in Calculate at calculate.c:21
        breakpoint already hit 1 time
(gdb) delete 1
(gdb)

```

Рис. 4.14: Запуск программы

14. При помощи утилиты split проанализировал коды файлов calculate.c и main.c. (рис. [4.15],[4.16],[4.17])

```

mamishonkov@dk8n53 ~ $ cd work
mamishonkov@dk8n53 ~/work $ cd os
mamishonkov@dk8n53 ~/work/os $ cd lab_prog
mamishonkov@dk8n53 ~/work/os/lab_prog $ splint calculate.c
Splint 3.1.2 --- 07 Dec 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:16:1: 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:22:1: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:28:1: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:34:1: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:35:4: 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:38:7: Return value type double does not match declared type float:
        (HUGE_VAL)
    To allow all numeric types to match, use +relaxtypes.
calculate.c:46:1: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:47:7: Return value type double does not match declared type float:
        (pow(Numeral, SecondNumeral))

```

Рис. 4.15: Анализ файла


```

calculate.c:50:7: Return value type double does not match declared type float:
    (sqrt(Numeral))
calculate.c:52:7: Return value type double does not match declared type float:
    (sin(Numeral))
calculate.c:54:7: Return value type double does not match declared type float:
    (cos(Numeral))
calculate.c:56:7: Return value type double does not match declared type float:
    (tan(Numeral))
calculate.c:60:7: Return value type double does not match declared type float:
    (HUGE_VAL)

Finished checking --- 15 code warnings
mamishonkov@dk8n53 ~/work/os/lab_prog $

```

Рис. 4.16: Анализ файла

```

mamishonkov@dk8n53 ~/work/os/lab_prog $ splint main.c
Splint 3.1.2 --- 07 Dec 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:16:3: Return value (type int) ignored: scanf("%s", Oper...

Finished checking --- 3 code warnings
mamishonkov@dk8n53 ~/work/os/lab_prog $

```

Рис. 4.17: Анализ файла

Ответы на контрольные вопросы:

1). Чтобы получить информацию о возможностях программ gcc, make, gdb и др. нужно воспользоваться командой тап или опцией -help(-h) для каждой команды.

2). Процесс разработки программного обеспечения обычно разделяется на следующие этапы:

планирование, включающее сбор и анализ требований к функционалу и другим характеристикам;
 проектирование, включающее в себя разработку базовых алгоритмов и спецификаций, с
 непосредственной разработкой приложения: кодирование – по сути создание исходного
 документирование. Для создания исходного текста программы разработчик может воспользо-

3). Для имени входного файла суффикс определяет какая компиляция требуется. Суффиксы указывают на тип объекта. Файлы с расширением (суффиксом) .с воспринимаются gcc как программы на языке C, файлы с расширением .с++ как файлы на языке C++, а файлы с расширением .о считаются объектными. Например, в команде «gcc-stdin.c»: gcc по расширению (суффиксу) .с распознает тип файла для компиляции и формирует объектный модуль – файл с расширением .о. Если требуется получить исполняемый файл с определённым именем (например, hello), то требуется воспользоваться опцией -o в качестве параметра задать имя создаваемого файла: «gcc-ohellomain.c». 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.п.с».

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 син-

таксис 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 = # Compile abcd.c normally abcd: abcd.c \$(CC) -o abcd \$(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 -c file.c -g. После этого для начала работы с gdb необходимо в командной строке ввести одноимённую команду, указав в качестве аргумента анализируемый бинарный файл: gdb file.o

8). Основные команды отладчика gdb:

backtrace – вывод на экран пути к текущей точке останова (по сути вывод – названий
break – установить точку останова (в качестве параметра может быть указан номер ст
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 выполнении;
`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 процессе выполнения данной лабораторной работы);
`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.
d. Более подробную информацию по работе с `gdb` можно получить с помощью команд `gdb` и `mangdb`.

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, `lint` – 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 Санализатор `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.

5 Выводы

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