

Шаблон отчёта по лабораторной работе № 14

Операционные Системы

Адебайо Ридвануллахи Айофе

Содержание

1	Цель работы	5
2	Задание	6
3	Выполнение лабораторной работы	9
4	Выводы	19
5	Контрольные вопросы	20

List of Tables

List of Figures

3.1	каталог	9
3.2	файлы	9
3.3	calculate.h файл	9
3.4	calculate.c файл	10
3.5	main.c файл	11
3.6	компиляция gcc	11
3.7	Makefile	12
3.8	gdb	13
3.9	run	13
3.10	list	14
3.11	list	14
3.12	list	15
3.13	list	15
3.14	breakpoints	16
3.15	run	16
3.16	print	16
3.17	display	17
3.18	delete	17
3.19	splint calculate.c	18
3.20	splint main.c	18

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. Реализация функций калькулятора в файле `calculate.h`:

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

```

3) == 0) return(tan(Numeral)); else { printf("Неправильно введено действие
"); return(HUGE_VAL); } } Интерфейсный файл calculate.h, описывающий
формат вызова функции калькулятора: ////////////////////////////////// //
calculate.h #ifndef CALCULATE_H_ #define CALCULATE_H_ float Calculate(float
Numerale, char Operation[4]); #endif /CALCULATE_H_/ Основной файл main.c,
реализующий интерфейс пользователя к калькулятору: //////////////////////////////////
// main.c

```

3. Выполните компиляцию программы посредством gcc: gcc -c calculate.c gcc -c main.c gcc calculate.o main.o -o calcul -lm
4. При необходимости исправьте синтаксические ошибки.
5. Создайте Makefile со следующим содержанием:

```

# # 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

```

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 точках останова: info breakpoints – Запустите программу внутри отладчика и убедитесь, что программа остановится в момент прохождения точки останова: run 5

- `backtrace` – Отладчик выдаст следующую информацию: `#0 Calculate (Numeral=5, Operation=0x7ffffffd280 "-") 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: `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 каталоге создал подкаталог ~/work/os/lab_prog.

```
raadebayjo@dk6n54 ~ $ mkdir work
mkdir: невозможно создать каталог «work»: Файл существует
raadebayjo@dk6n54 ~ $ mkdir work/os
mkdir: невозможно создать каталог «work/os»: Файл существует
raadebayjo@dk6n54 ~ $ mkdir work/os/lab_prog
raadebayjo@dk6n54 ~ $ cd work/os/lab_prog
raadebayjo@dk6n54 ~/work/os/lab_prog $
```

Figure 3.1: каталог

2. Создал в нём файлы: calculate.h, calculate.c, main.c.

```
raadebayjo@dk6n54 ~/work/os/lab_prog $ touch calculate.h calculate.c main.c
raadebayjo@dk6n54 ~/work/os/lab_prog $
```

Figure 3.2: файлы

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

Figure 3.3: calculate.h файл

```

1 #include <stdio.h>
2 #include <math.h>
3 #include <string.h>
4 #include "calculate.h"
5 float
6 Calculate(float Numeral, char Operation[4])
7 {
8     float SecondNumeral;
9     if(strncmp(Operation, "+", 1) == 0)
10    {
11        printf("Второе слагаемое: ");
12        scanf("%f",&SecondNumeral);
13        return(Numeral + SecondNumeral);
14    }
15    else if(strncmp(Operation, "-", 1) == 0)
16    {
17        printf("Вычитаемое: ");
18        scanf("%f",&SecondNumeral);
19        return(Numeral - SecondNumeral);
20    }
21    else if(strncmp(Operation, "*", 1) == 0)
22    {
23        printf("Множитель: ");
24        scanf("%f",&SecondNumeral);
25        return(Numeral * SecondNumeral);
26    }
27    else if(strncmp(Operation, "/", 1) == 0)
28    {
29        printf("Делитель: ");
30        scanf("%f",&SecondNumeral);
31        if(SecondNumeral == 0)
32        {

```

Figure 3.4: calculate.c файл

```

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

```

Figure 3.5: main.c файл

3. Выполнил компиляцию программы посредством gcc

```

raadebayjo@dk6n54 ~/work/os/lab_prog $ gcc -c calculate.c
raadebayjo@dk6n54 ~/work/os/lab_prog $ gcc -c main.c
raadebayjo@dk6n54 ~/work/os/lab_prog $ gcc calculate.o main.o -o calcul -lm
raadebayjo@dk6n54 ~/work/os/lab_prog $

```

Figure 3.6: компиляция gcc

4. Исправил синтаксические ошибки в файле main.c (удалил & перед operator в линии scanf("%s",&Operation);)
5. Создал Makefile со следующим содержанием

```
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
```

Figure 3.7: Makefile

6.С помощью gdb выполнил отладку программы calcul (перед использованием gdb исправьте Makefile)

- Запустил отладчик GDB, загрузил в него программу для отладки: `gdb ./calcul`

```

raadebayjo@dk6n54 ~/work/os/lab_prog $ gdb ./calcul
GNU gdb (Gentoo 10.1 vanilla) 10.1
Copyright (C) 2020 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) █

```

Figure 3.8: gdb

- Для запуска программы внутри отладчика ввел команду run: run

```

(gdb) run
Starting program: /afs/.dk.sci.pfu.edu.ru/home/r/a/raadebayjo
calcul
Число: 8
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): -
Вычитаемое: 5
    3.00
[Inferior 1 (process 23745) exited normally]
(gdb) █

```

Figure 3.9: run

- Для постраничного (по 9 строк) просмотра исходного код использовал команду list: list

```

(gdb) list
1      #include <stdio.h>
2      #include "calculate.h"
3
4      int main (void)
5      {
6          float Numeral;
7          char Operation[4];
8          float Result;
9          printf("Число: ");
10         scanf("%f",&Numeral);
(gdb)

```

Figure 3.10: list

- Для просмотра строк с 12 по 15 основного файла использовал list с параметрами: list 12,15

```

(gdb) list 12,15
12         scanf("%s",Operation);
13         Result = Calculate(Numeral, Operation);
14         printf("%6.2f\n",Result);
15         return 0;
(gdb)

```

Figure 3.11: list

- Для просмотра определённых строк не основного файла использовал list с параметрами: list calculate.c:20,29


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

Figure 3.12: list

- Установил точку останова в файле calculate.c на строке номер 21: list calculate.c:20,27 break 21

```
(gdb) list calculate.c:20,27
20         }
21         else if(strncmp(Operation, "*", 1) == 0)
22         {
23             printf("Множитель: ");
24             scanf("%f",&SecondNumeral);
25             return(Numeral * SecondNumeral);
26         }
27         else if(strncmp(Operation, "/", 1) == 0)
(gdb) break21
Undefined command: "break21". Try "help".
(gdb) break 21
Breakpoint 1 at 0x991: file calculate.c, line 21.
(gdb) █
```

Figure 3.13: list

- Вывел информацию об имеющихся в проекте точка останова: info breakpoints

```
(gdb) info breakpoints
Num      Type      Disp Enb Address              What
1        breakpoint keep y   0x0000000000000099 in Calculate
                                at calculate.c:21
(gdb)
```

Figure 3.14: breakpoints

- Запустил программу внутри отладчика и убедитесь, что программа остановится в момент прохождения точки останова: run 5
- backtrace

```
(gdb) run
Starting program: /afs/.dk.sci.pfu.edu.ru/home/r/a/raadebayjo/work/os/lab_prog
calcul
Число: 5
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): *
Breakpoint 1, Calculate (Numeral=5, Operation=0x7fffffffcf34 "*")
  at calculate.c:21
21      else if(strncmp(Operation, "*", 1) == 0)
(gdb) backtrace
#0 Calculate (Numeral=5, Operation=0x7fffffffcf34 "*") at calculate.c:21
#1 0x0000555553400c3f in main () at main.c:13
(gdb)
```

Figure 3.15: run

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

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

Figure 3.16: print

- Сравнил с результатом вывода на экран после использования команды:
display Numeral


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

Figure 3.17: display

- Убрал точки останова: info breakpoints delete 1

```
(gdb) info breakpoints  
Num    Type           Disp Enb Address                  What  
1      breakpoint      keep y   0x0000555555400901 in Calculate  
                                at calculate.c:21  
        breakpoint already hit 1 time  
(gdb) delete 1
```

Figure 3.18: delete

7. С помощью утилиты splint попробовал проанализировать коды файлов calculate.c и main.c.

```
Терминал - raadebayjo@dk6n65:~/work/os/lab_prog
Файл  Правка  Вид  Терминал  Вкладки  Справка
raadebayjo@dk6n65 ~/work/os/lab_prog $ splint calculate.c
Splint 3.1.2 --- 13 Jan 2021

calculate.h:4:37: 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:6:31: Function parameter Operation declared as manifest array (size
                    constant is meaningless)
calculate.c: (in function Calculate)
calculate.c:12:3: 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:18:3: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:24:3: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:30:3: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:31:6: 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:34:10: Return value type double does not match declared type float:
```

Figure 3.19: splint calculate.c

```
raadebayjo@dk6n65 ~/work/os/lab_prog $ splint main.c
Splint 3.1.2 --- 13 Jan 2021

calculate.h:4:37: 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:10:2: 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:12:2: Return value (type int) ignored: scanf("%s", Oper...

Finished checking --- 3 code warnings
raadebayjo@dk6n65 ~/work/os/lab_prog $
```

Figure 3.20: splint main.c

4 Выводы

В результате работы , я приобрёл простейшие навыки разработки, анализа, тестирования и отладки приложений в Линукс

5 Контрольные вопросы

1. Дополнительную информацию о 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. `makefile` для программы `abcd.c` мог бы иметь вид:

6

7

Makefile

8

```
CC = gcc
CFLAGS =
LIBS = -lm
calcul: calculate.o main.o
gcc calculate.o main.o -o calcul $(LIBS)
calculate.o: calculate.c calculate.h
gcc -c calculate.c $(CFLAGS)
main.o: main.c calculate.h
gcc -c main.c $(CFLAGS)
clean: -rm calcul .o ~
# End Makefile
```

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

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

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

– break – устанавливает точку останова; параметром может быть номер строки

или название

функции;

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

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

– delete – удаляет точку останова или контрольное выражение;

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

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

– info breakpoints – выводит список всех имеющихся точек останова;

– info watchpoints – выводит список всех имеющихся контрольных выражений;

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

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

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

– run – запускает программу на выполнение;

– set – устанавливает новое значение переменной

– step – пошаговое выполнение программы;

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

9. 1) Выполнили компиляцию программы 2) Увидели ошибки в программе 3) Открыли редактор и исправили программу 4) Загрузили программу в отладчик gdb 5) run — отладчик выполнил программу, мы ввели требуемые значения. 6) программа завершена, gdb не видит ошибок.

10. 1 и 2.) Мы действительно забыли закрыть комментарии; 3.) 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:
- cscope - исследование функций, содержащихся в программе;
 - splint — критическая проверка программ, написанных на языке Си.
12. 1. Проверка корректности задания аргументов всех исп функций, а также типов возвращаемых ими значений;
2. Поиск фрагментов исходного текста, корректных с точки зрения синтаксиса языка Си, но малоэффективных с точки зрения их реализации или содержащих в себе семантические ошибки;
3. Общая оценка мобильности пользовательской программы.