

Лабораторная работа №14

По дисциплине Операционные системы

Выполнил Гамаюнов Н.Е., студент ФФМиЕН РУДН, НПМбд-01-20, 1032201717

Преподаватель Кулябов Дмитрий Сергеевич

Москва, 2021 г.

Цель работы

Приобрести простейшие навыки разработки, анализа, тестирования и отладки приложений в ОС типа UNIX/Linux на примере создания на языке программирования C калькулятора с простейшими функциями.

Задания

1. Создать файлы для корректной работы калькулятора
2. Изучить Makefile, исправить имеющийся шаблон
3. На практике воспользоваться gcc для отладки приложения
4. Проверить код с помощью splint

Выполнение лабораторной работы

1. В домашнем каталоге создал подкаталог `~/work/os/lab_prog` (рисунок 1)

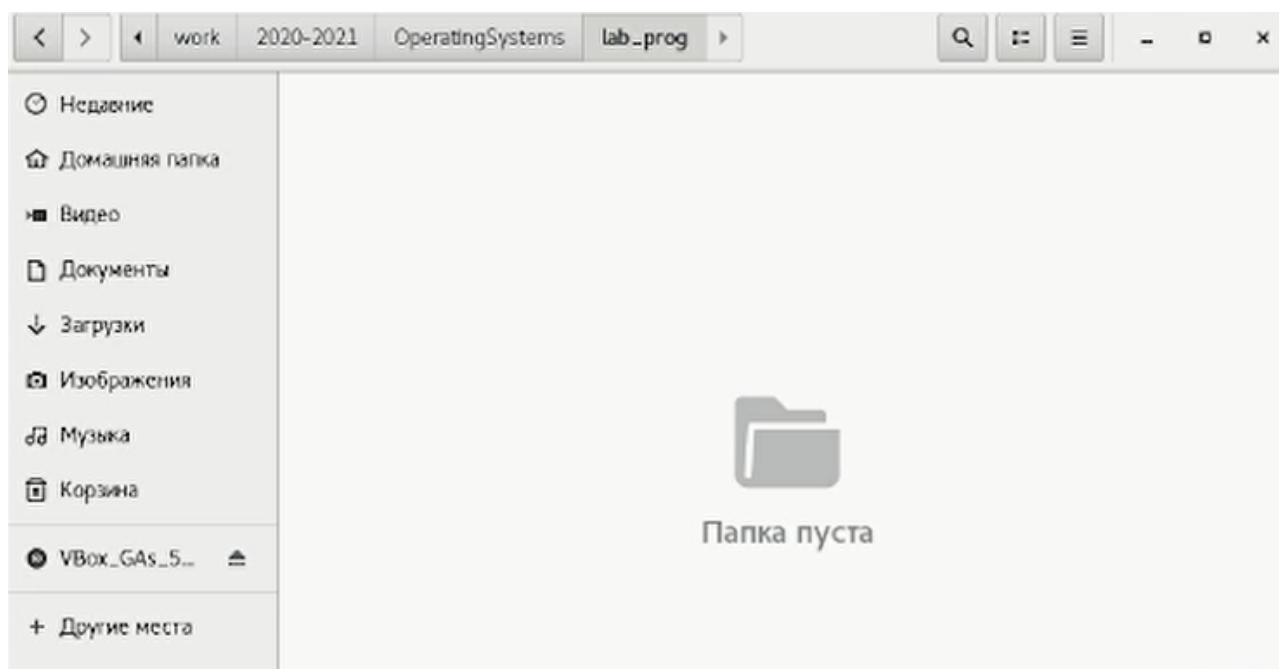


Рисунок 1.

2. Создал в нём файлы: `calculate.h`, `calculate.c`, `main.c` (рисунок 2)

```
[negamayunov@negamayunov lab_prog]$ touch calculate.h; touch calculate.c; touch calculate.c  
[negamayunov@negamayunov lab_prog]$ touch main.c
```

Рисунок 2.

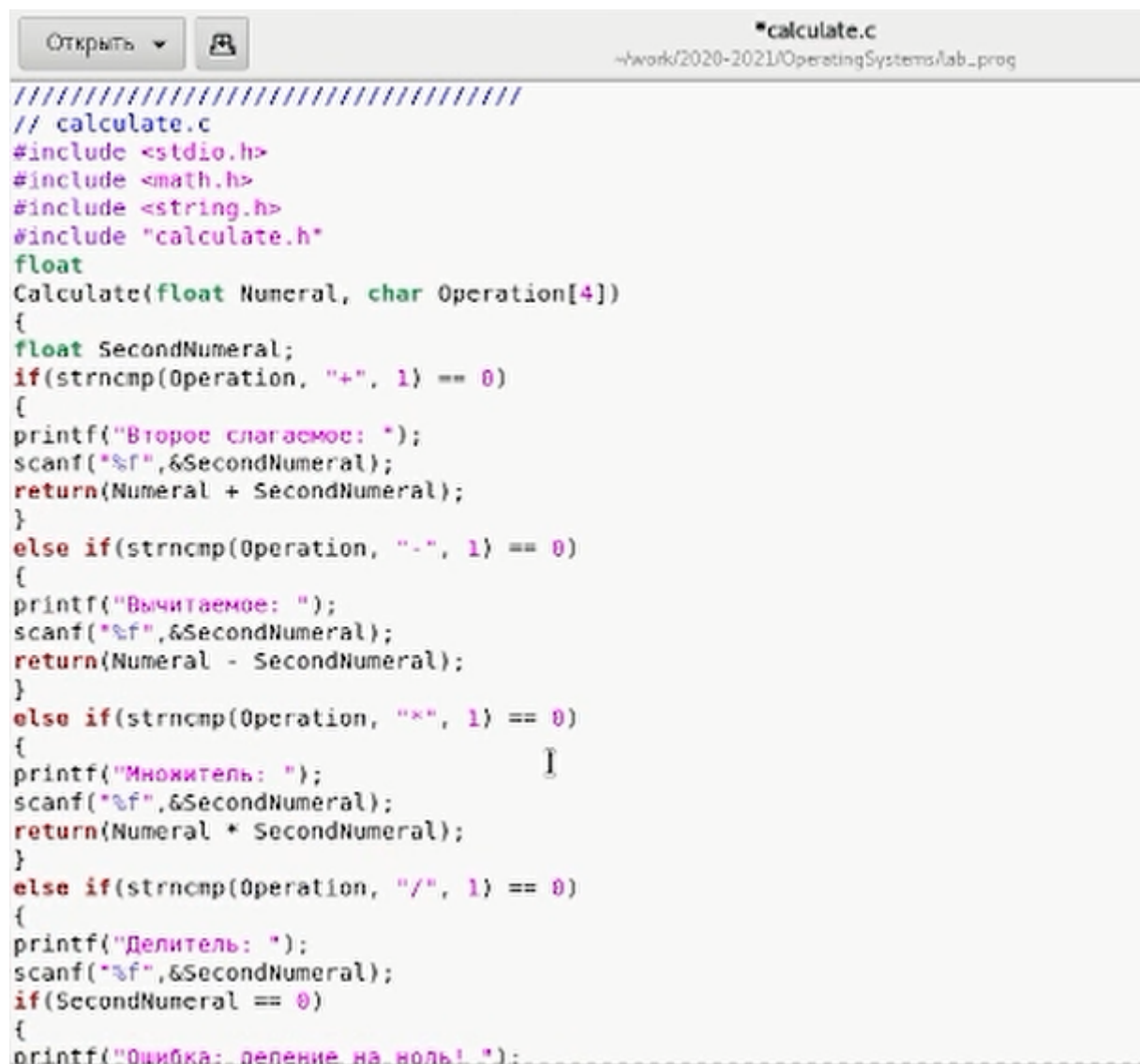
Это будет примитивнейший калькулятор, способный складывать, вычитать, умножать и делить, возводить число в степень, брать квадратный корень, вычислять `sin`, `cos`, `tan`. При запуске он

будет запрашивать первое число, операцию, второе число. После этого программа выведет результат и остановится.

Реализация функций калькулятора в файле `calculate.c` - Рисунок 3

Интерфейсный файл `calculate.h`, описывающий формат вызова функции калькулятора - Рисунок 5.

Основной файл `main.c`, реализующий интерфейс пользователя к калькулятору - Рисунок 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);
        return(Numeral * SecondNumeral);
    }
    else if(strncmp(Operation, "/", 1) == 0)
    {
        printf("Делитель: ");
        scanf("%f", &SecondNumeral);
        if(SecondNumeral == 0)
        {
            printf("Ошибка: деление на ноль! ");
        }
    }
}

```

Рисунок 3.



```

main.c [изменённые] — KWrite
Файл  Правка  Вид  Закладки  Сервис  Настройка  Справка
Создать  Открыть  Сохранить  Сохранить как  Закрыть  Отменить действие

////////////////////////////////////
// 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;
}

```

Рисунок 4.



```

*calculate.h
~/work/2020-2021/OperatingSystems/lab_prog
Сохранить

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

```

Рисунок 5.

3. Выполнил компиляцию посредством gcc (рисунок 6):

```

[negamayunov@negamayunov ~]$ cd work/2020-2021/OperatingSystems/lab_prog
[negamayunov@negamayunov lab_prog]$ gcc -c calculate.c
[negamayunov@negamayunov lab_prog]$ gcc -c main.c
[negamayunov@negamayunov lab_prog]$ gcc calculate.o main.o -o calcul -lm

```

Рисунок 6.

4. Синтаксических ошибок компилятор не обнаружил.

5. Создал Makefile (рисунок 7), отредактировал его (рисунок 8)

```

negamayunov@negamayunov:~/work/2020-2021/OperatingSystems/lab_prog
Файл Правка Вид Поиск Терминал Справка
#
## 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

```

Рисунок 7.

```

#
## 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

```

Рисунок 8.

Переменные CC, CFLAGS и LIBS, Заданные в начале, нужны, чтобы облегчить написание программы, - вместо gcc я могу в любом месте просто написать \$CC.

Далее описываются цели: calcul, calculate.o, main.o и clean

- после выполнения `make calcul` мы получим исполняемый файл calcul
- calculate.o и main.o позволяют получить объектные файлы
- clean удаляет ненужные объектные файлы

6. С помощью gdb выполнил отладку программы calcul (рисунки 9-12):

Запустил отладчик GDB, загрузив в него программу для отладки:

```
gdb ./calcul
```

- Для запуска программы внутри отладчика ввел команду run: run
- Для постраничного (по 9 строк) просмотра исходного код использовал команду list
- Для просмотра строк с 12 по 15 основного файла использовал list с параметрами: `list 12,15`
- Для просмотра определённых строк не основного файла использовал list с параметрами: `list calculate.c:20,29`
- Установил точку останова в файле calculate.c на строке номер 21:

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

- Вывел информацию об имеющихся в проекте точка останова: `info breakpoints`
- Запустил программу внутри отладчика и убедился, что программа остановится в момент прохождения точки останова:

```
run  
5  
-  
backtrace
```

- Посмотрел, чему равно на этом этапе значение переменной Numeral, введя: `print Numeral`
- Сравнил с результатом вывода на экран после использования команды: `display Numeral`
print выводит название переменной непосредственно из кода программы, а display имеет более понятную форму вывода.
- Убрал точки останова:
`info breakpoints delete 1`

```
(gdb) run
Starting program: /home/neganayunov/work/2020-2021/OperatingSystems/lab_
alcul
Число: 4
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): -
Вычитаемое: 5
-1.00
[Inferior 1 (process 3817) exited normally]
Missing separate debuginfos, use: debuginfo-install glibc-2.17-317.el7.x
(gdb) list
2      // main.c
3      #include <stdio.h>
4      #include "calculate.h"
5      int
6      main (void)
7      {
8      float Numeral;
9      char Operation[4];
10     float Result;
11     printf("Число: ");
(gdb) █
```

Рисунок 9.

```
(gdb) list 12,15
12     scanf("%f",&Numeral);
13     printf("Операция (+,-,*,/,pow,sqrt,sin,cos,tan): ");
14     scanf("%s",&Operation);
15     Result = Calculate(Numeral, Operation);
(gdb) list calculate.c:20,29
20     scanf("%f",&SecondNumeral);
21     return(Numeral - SecondNumeral);
22 }
23 else if(strncmp(Operation, "+", 1) == 0)
24 {
25     printf("Множитель: ");
26     scanf("%f",&SecondNumeral);
27     return(Numeral * SecondNumeral);
28 }
29 _ else if(strncmp(Operation, "/", 1) == 0)
```

Рисунок 10.

```
(gdb) break 21
Breakpoint 1 at 0x4007fd: file calculate.c, line 21.
(gdb) info breakpoints
Num    Type             Disp Enb Address                  What
1      breakpoint       keep y   0x00000000004007fd in Calculate
                                           at calculate.c:21
(gdb) run
Starting program: /home/neganayunov/work/2020-2021/OperatingSystems/lab_
alcul
Число: 5
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): -
Вычитаемое: backtrace
Breakpoint 1, Calculate (Numeral=5, Operation=0x7fffffffdd40 "-")
at calculate.c:21
21     return(Numeral - SecondNumeral);
(gdb) █
```

Рисунок 11.


```
(gdb) print Numeral
$1 = 5
(gdb) display Numeral
1: Numeral = 5
(gdb) info breakpoints
Num    Type             Disp Enb Address                  What
1      breakpoint      keep y   0x00000000004007fd in Calculate
                                           at calculate.c:21
breakpoint already hit 1 time
(gdb) delete 1
```

Рисунок 12.

7. С помощью утилиты splint проанализировал коды файлов calculate.c и main.c (рисунок 13). В каждой программе нашлись ошибки, в основном, это были проблемы с несовпадением возвращаемого значения и значения, заданного при инициализации (например, когда переменной типа double присваивается значение float)

```
SecondNumeral == 0
Two real (float, double, or long double) values are compared directly
== or != primitive. This may produce unexpected results since floating
representations are inexact. Instead, compare the difference to FLT_EPSILON
or DBL_EPSILON. (Use -realcompare to inhibit warning)
calculate.c:36:7: Return value type double does not match declared type
(HUGE_VAL)
To allow all numeric types to match, use +relaxtypes.
calculate.c:44:1: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:45:7: Return value type double does not match declared type
(pow(Numeral, SecondNumeral))
calculate.c:48:7: Return value type double does not match declared type
(sqrt(Numeral))
calculate.c:50:7: Return value type double does not match declared type
(sin(Numeral))
calculate.c:52:7: Return value type double does not match declared type
(cos(Numeral))
calculate.c:54:7: Return value type double does not match declared type
(tan(Numeral))
calculate.c:58:7: Return value type double does not match declared type
(HUGE_VAL)

Finished checking --- 15 code warnings
[negamayunov@negamayunov lab_prog]$ splint main.c
```

Рисунок 13.

Выводы

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

Контрольные вопросы

1. С помощью команды man.
2. 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.
3. Суффикс - это расширение. Например, 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
4. Не только компиляция программы, но и получение 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:

```
<цель_1> <цель_2> ... : <зависимость_1>    <зависимость_2>    ...  
<команда 1>  
...  
<команда n>
```

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.

Рассмотрим пример Makefile для написанной выше простейшей программы, выводящей на экран приветствие 'Hello World!':

```
hello: main.c  
    gcc -o hello main.c
```

Здесь в первой строке `hello` — цель, `main.c` — название файла, который мы хотим скомпилировать; во второй строке, начиная с табуляции, задана команда компиляции gcc с

опциями. Для запуска программы необходимо в командной строке набрать команду make:

```
make
```

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.

7. 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
```

Затем можно использовать по мере необходимости различные команды gdb.

8. Некоторые команды 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 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. Запустил отладчик GDB, загрузив в него программу для отладки:

```
gdb ./calcul
```

- Для запуска программы внутри отладчика ввел команду `run`: `run`
- Для постраничного (по 9 строк) просмотра исходного код использовал команду `list`
- Для просмотра строк с 12 по 15 основного файла использовал `list` с параметрами: `list 12,15`
- Для просмотра определённых строк не основного файла использовал `list` с параметрами: `list calculate.c:20,29`
- Установил точку останова в файле `calculate.c` на строке номер 21:

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

- Вывел информацию об имеющихся в проекте точка останова: `info breakpoints`
- Запустил программу внутри отладчика и убедился, что программа остановится в момент прохождения точки останова:

```
run  
5  
-  
backtrace
```

- Посмотрел, чему равно на этом этапе значение переменной `Numeral`, введя: `print Numeral`
- Сравнил с результатом вывода на экран после использования команды: `display Numeral`

- Убрал точки останова:

```
info breakpoints  
delete 1
```

10. Мой компилятор никак не отреагировал на ошибку в коде (см. скринкаст к работе): в строке `scanf("%s", &Operation);` не нужен был знак амперсанда `&`.
11. Например, с помощью утилиты `splint` можно проверить исходный код на ошибки.
12. Эта утилита анализирует программный код, проверяет 74 Лабораторная работа № 11. Средства, применяемые при разработке программного... корректность задания аргументов использованных в программе функций и типоввозвращаемых значений, обнаруживает синтаксические и семантические ошибки.

Источник всей информации, которой я пользовался для ответа на вопросы и выполнения работы
- [Методические рекомендации к лабораторной работе №14](#)

Библиография

- [Кулябов Д.С. и др. Операционные системы. Методические рекомендации к лабораторной работе №14](#)