# Отчёт по лабораторной работе №13

***дисциплина: Операционные системы***

Максим Александрович Мишонков

# Содержание

# Список иллюстраций

# 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 C 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])

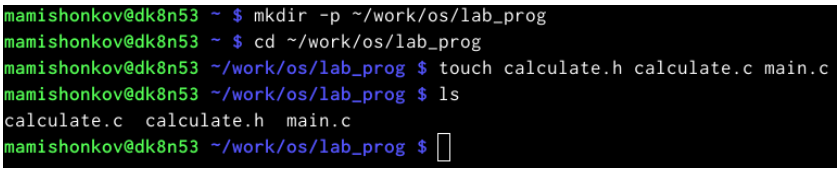


Рис. 4.1: Создание каталога и файлов

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

```c
////////////////////////////////
// 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("%6.2f\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 cakculate.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])

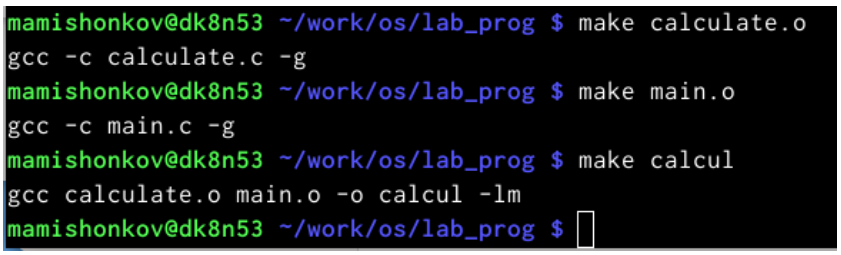


Рис. 4.10: Компиляция файлов

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

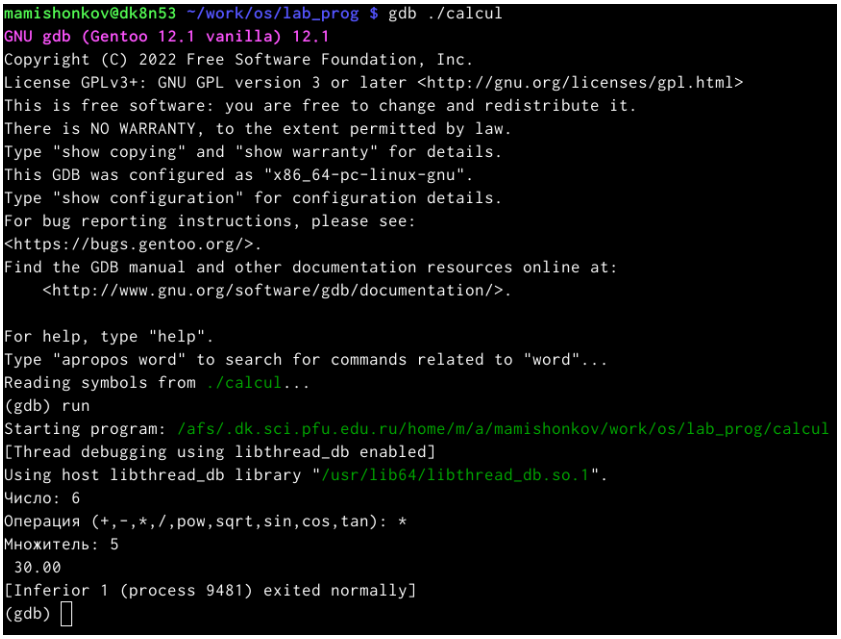


Рис. 4.11: Отладчик

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


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 выведения информации об имеющихся точек останова). (рис. [4.13])


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])

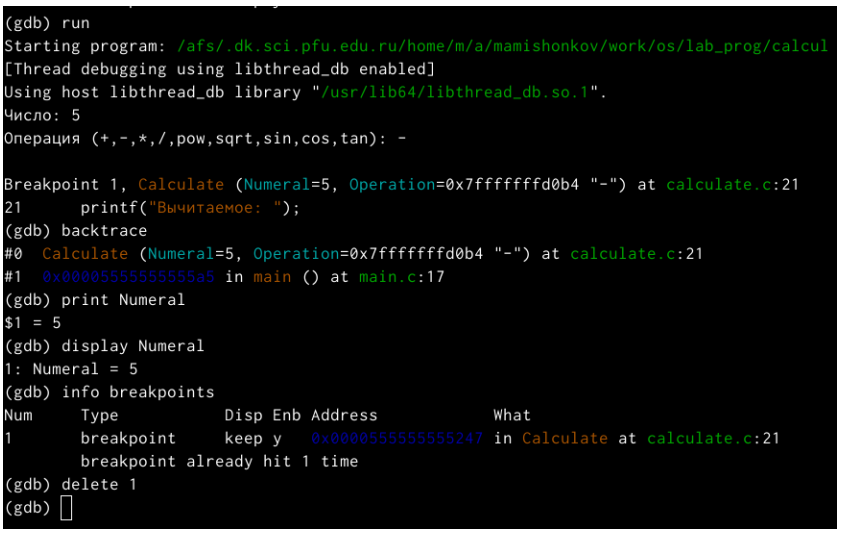


Рис. 4.14: Запуск программы

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

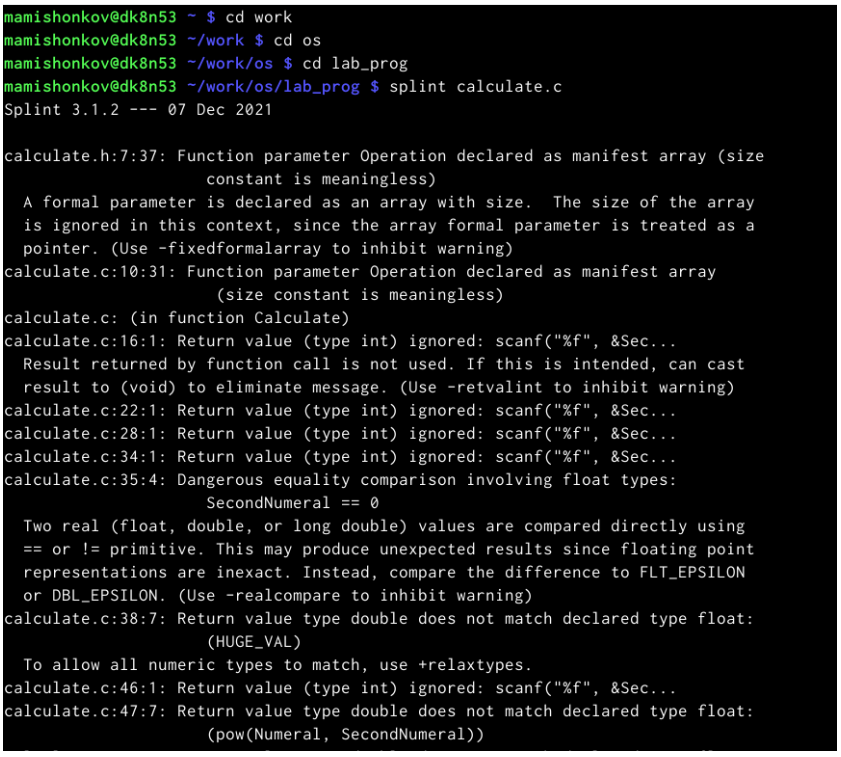


Рис. 4.15: Анализ файла

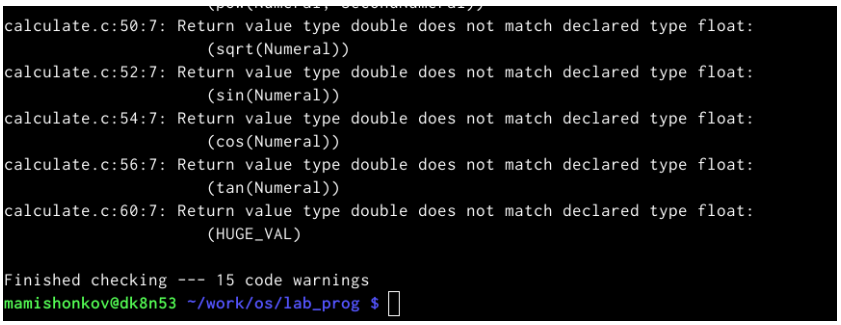


Рис. 4.16: Анализ файла

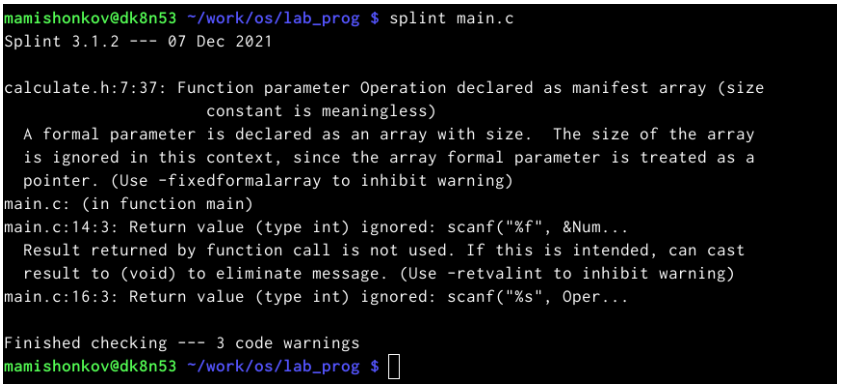


Рис. 4.17: Аналих файла

**Ответы на контрольные вопросы:**

1). Чтобы получить информацию о возможностях программ gcc, make, gdbи др.нужно воспользоваться командой manили 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 характер
проектирование, включающее в себя разработку базовых алгоритмов и спецификаций, с
непосредственная разработка приложения: окодирование –по сути создание исходного
документирование. Для создания исходного текста программы разработчик может воспо

3). Для имени входного файла суффикс определяет какая компиляция требуется. Суффиксы указывают на тип объекта. Файлы с расширением (суффиксом) .своспринимаются gcскак программы на языке C, файлы с расширением .ссили .C–как файлы на языке C++, а файлы срасширением .осчитаются объектными.Например, в команде «gcc-cmain.c»:gccпо расширению (суффиксу) .сраспознает тип файла для компиляции и формирует объектный модуль –файл с расширением .o. Если требуется получить исполняемый файл с определённым именем (например, hello), то требуется воспользоваться опцией -oи в качестве параметра задать имя создаваемого файла: «gcc-ohellomaiВ ходе выполнения данной лабораторной работы я приобрелапростейшие навыки разработки, анализа, тестирования и отладки приложений в ОС типа UNIX/Linuxна примере создания на языке программирования C калькулятора с простейшими функциями.n.c».

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 makeнеобходимо в корне рабочего каталога с Вашим проектом создать файл с названием makefileили 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 Makefileможет 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 Makefileимеет 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 = gccCFLAGS =# Compile abcd.c normalyabcd: abcd.c$(CC) -o abcd $(CFLAGS) abcd.cclean:-rm abcd .*o* ~# EndMakefileforabcd.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 cleanпроизводит 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 GNUдля ОС типа UNIXвходит отладчик GDB(GNUDebugger). Для использования GDB необходимо скомпилировать анализируемый код программы таким образом, чтобы отладочная информация содержалась в результирующем бинарном файле. Для этого следует воспользоваться опцией -gкомпилятора gcc: gcc-cfile.c-g. После этого для начала работы с gdbнеобходимо в командной строке ввести одноимённую команду, указав в качестве аргумента анализируемый бинарный файл: gdbfile.o

8). Основные команды отладчика gdb:

```
backtrace – вывод на экран пути к текущей точке останова (по сутивывод – названий
break – установить точку останова (в качестве параметра можетбыть 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 пр

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 (вВ ходе выполнения данной лабораторной рабо

next – выполнить программу пошагово, но без выполнения вызываемых в программе фун

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 програм

d. Более подробную информацию по работе с gdb можно получить с помощью команд gdb

hи 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 массивасимволов уже является указателемна первый элементэтого 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 splintгенерирует 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 работt программы, переменные с некорректно заданными значениямии 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 C калькулятора с простейшими функциями.