

РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ

Факультет физико-математических и естественных наук

Кафедра прикладной информатики и теории вероятностей

ОТЧЕТ

ПО ЛАБОРАТОРНОЙ РАБОТЕ № 14

дисциплина: Операционные системы

Студент: Соболевский Денис Андреевич

Группа: НФИбд-02-20

Преподаватель: Велиева Татьяна Рефатовна

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.

Задачи:

1. Научиться выполнять компиляцию по средствам командной строки;
2. Освоить отладчик GDB;
3. Научиться анализировать исходные коды.

Теоретическое введение:

Этапы разработки приложений

- 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.

Компиляция исходного текста и построение исполняемого файла

Для компиляции, например, файла `main.c` используют команду:

```
gcc -c main.c
```

Если требуется получить исполняемый файл с определённым именем (например, `hello`), то требуется воспользоваться опцией `-o` и в качестве параметра задать имя создаваемого файла:

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

С прочими опциями компилятора `gcc` можно ознакомиться в статье ["Опции компиляторов"](#)^[1].

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.

Тестирование и отладка

Для использования отладчика 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.

Анализ исходного текста программы

Для анализа кода программы example.c следует выполнить следующую команду:

```
splint example.c
```

В ходе работы над понадобится установить утилиту splint на Centos 7, поэтому воспользуемся статьёй "[УСТАНОВКА ПАКЕТОВ В CENTOS 7](#)"^[2].

Все коды, которые использовались во время выполнения работы были взяты из *Лабораторной работы №14*^[3].

Выполнение работы:

1, 2. В домашнем каталоге нам нужно создать подкаталог ~/work/os/lab_prog.

Создаём каталог os в уже созданном ранее каталоге work и создаём в нём подкаталог lab_prog - `mkdir lab_prog` (рисунки 1). Перейдём в него командой `cd lab_prog`. Создадим в нём файлы: calculate.h, calculate.c, main.c с помощью текстового редактора emacs (рисунки 1). Это будет примитивнейший калькулятор.

```
[dasobolevskiy@dasobolevskiy lab_prog]$ emacs calculate.h
[dasobolevskiy@dasobolevskiy lab_prog]$ emacs calculate.c
[dasobolevskiy@dasobolevskiy lab_prog]$ emacs main.c
```

рисунок 1: создание подкаталога ~/work/os/lab_prog и файлов calculate.h, calculate.c, main.c

В файл calculate.h вводим код на языке программирования C, предоставленный в материалах к ЛР №14 (рисунки 2)

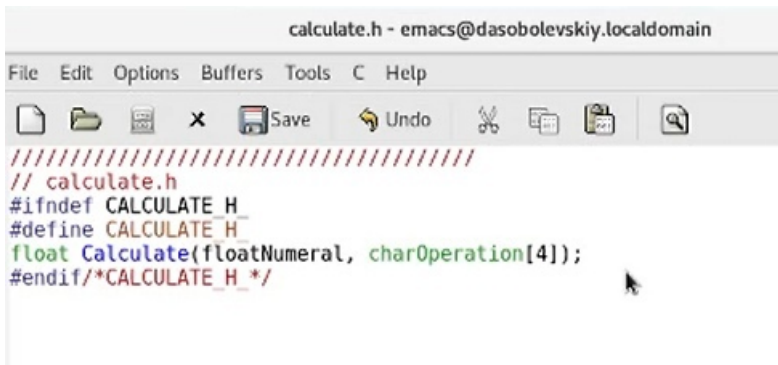


рисунок 2: файл calculate.h

То же делаем с файлами calculate.c (рисунки 3) и main.c (рисунки 4).

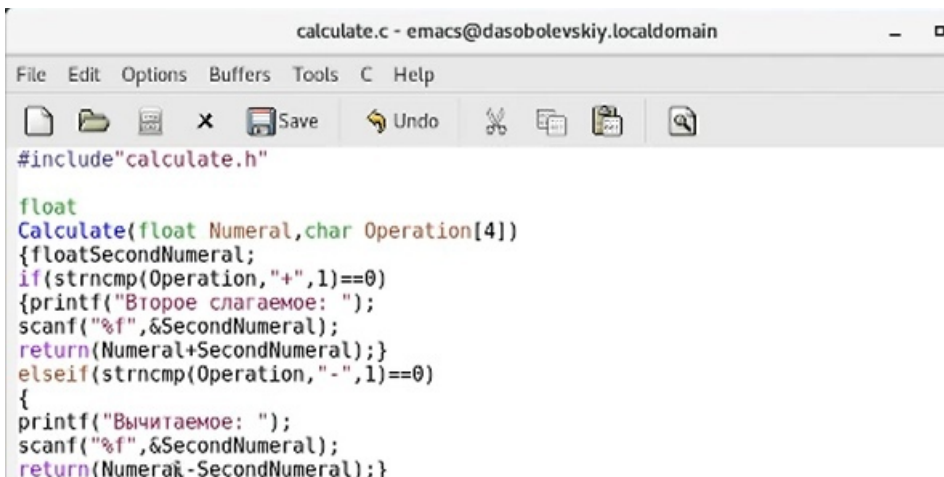
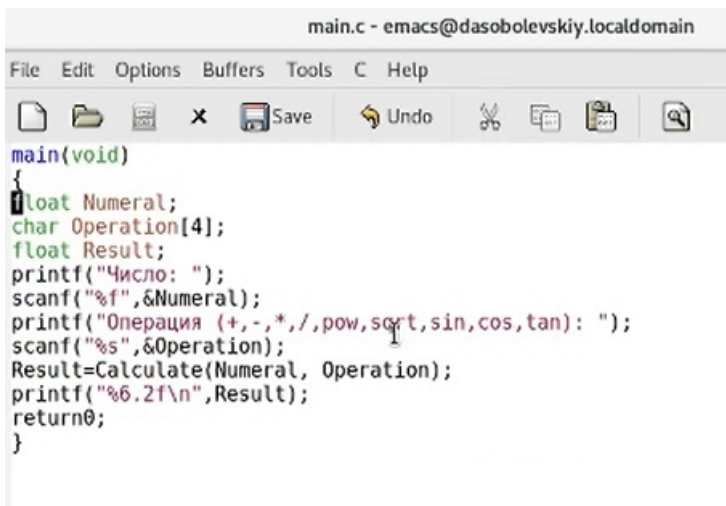


рисунок 3: файл calculate.c

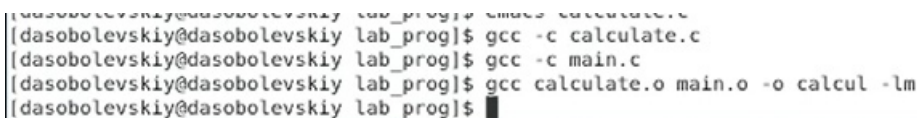


```
main.c - emacs@dasobolevskiy.localdomain
File Edit Options Buffers Tools C Help
[Icons] Save Undo [Icons]
main(void)
{
    float Numeral;
    char Operation[4];
    float Result;
    printf("Число: ");
    scanf("%f",&Numeral);
    printf("Операция (+,-,*,/,pow,sqrt,sin,cos,tan): ");
    scanf("%s",&Operation);
    Result=Calculate(Numeral, Operation);
    printf("%.2f\n",Result);
    return 0;
}
```

рисунок 4: файл main.c

3, 4. Теперь выполним компиляцию программы посредством gcc, ввода следующие команды (рисунок 5):

```
gcc -c calculate.c
gcc -c main.c
gcc calculate.o main.o -o calcul -lm
```

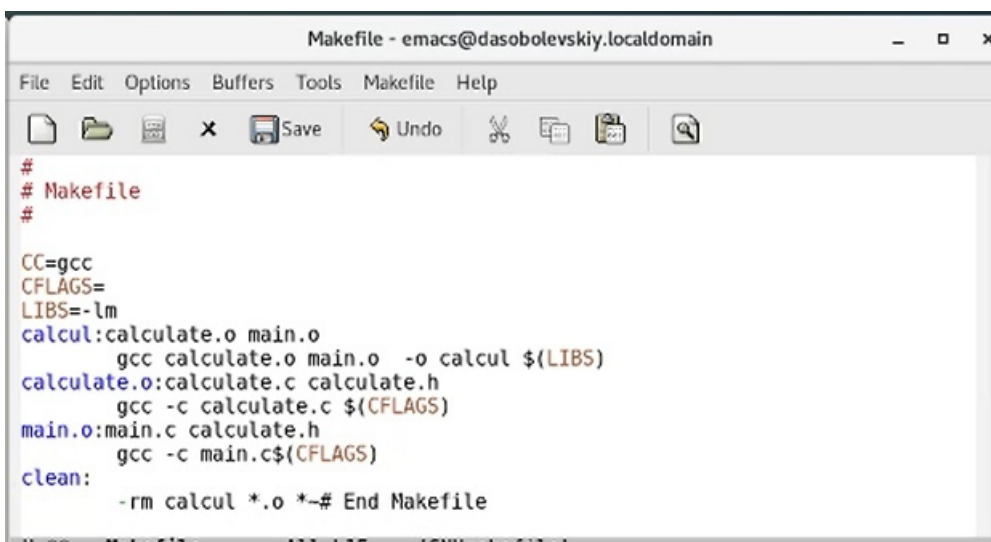


```
dasobolevskiy@dasobolevskiy lab_prog$ gcc -c calculate.c
[dasobolevskiy@dasobolevskiy lab_prog]$ gcc -c main.c
[dasobolevskiy@dasobolevskiy lab_prog]$ gcc calculate.o main.o -o calcul -lm
[dasobolevskiy@dasobolevskiy lab_prog]$
```

рисунок 5: компиляция программы посредством gcc

Видим, что система не выдает нам сообщений об ошибках, следовательно код написан правильно, и нам нечего исправлять.

5. Создадим Makefile, который будет расположен в каталоге lab_prog, поскольку makefile должен находиться в том же месте, где и проект, связанный с ним. Создаем файл с помощью редактора emacs и вводим в него предложенный код файла из лабораторной работы (рисунок 6).



```
Makefile - emacs@dasobolevskiy.localdomain
File Edit Options Buffers Tools Makefile Help
[Icons] Save Undo [Icons]
#
# 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 6: создание Makefile

6. С помощью gdb выполним отладку программы calcul. Для использования GDB нам необходимо сначала скомпилировать анализируемый код программы таким образом, чтобы отладочная информация содержалась в результирующем бинарном файле (рисунок 7). Для этого следует воспользоваться опцией -g компилятора gcc, тогда синтаксис компиляции будет следующим:

```
gcc -c [имя файла] -g
```

```
[dasobolevskiy@dasobolevskiy lab_prog]$ gcc -c calculate.c -g
[dasobolevskiy@dasobolevskiy lab_prog]$ gcc -c main.c -g
[dasobolevskiy@dasobolevskiy lab_prog]$ gcc calculate.o main.o -o calcul -lm
[dasobolevskiy@dasobolevskiy lab_prog]$
```

рисунок 7: компиляция перед запуском GDB

- Запустим отладчик GDB, загрузив в него программу для отладки: `gdb ./calcul` (рисунок 8).

```
[dasobolevskiy@dasobolevskiy lab_prog]$ gdb ./calcul
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-120.el7
Copyright (C) 2013 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-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/dasobolevskiy/work/os/lab_prog/calcul...done.
(gdb)
```

рисунок 8: запуск отладчика GDB

- Теперь нам нужно запустить программу внутри отладчика. Для этого внутри отладчика введем команду `run` (рисунок 9).

```
(gdb) run
Starting program: /home/dasobolevskiy/work/os/lab_prog/./calcul
Число: 4
Операция (+, -, *, /, pow, sqrt, sin, cos, tan): +
Второе слагаемое: 5
9.00
[Inferior 1 (process 4573) exited normally]
Missing separate debuginfos, use: debuginfo-install glibc-2.17-317.el7.x86_64
(gdb)
```

рисунок 9: запуск программы в отладчике

Видим, что программа была успешно запущена. На ввод нам предлагается ввести какое-либо число (вводим 4), операцию, которая будет производится с ним (в нашем случае это сложение), далее нам предлагается ввести второе слагаемое (5). Результат выделен черным - 9. Программа работает исправно.

- Для постраничного (по 9 строк) просмотра исходного код используем команду `list` (рисунок 10). Видим, что действительно вывелось 9 строк (4-13).

```
(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: ");
```

рисунок 10: постраничный вывод list

- Для просмотра строк с 12 по 15 основного файла используем list с параметрами - `list 12,15` (рисунок 11).

```
(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)
```

рисунок 11: просмотр определенных строк

- Для просмотра определённых строк не основного файла используем list с параметрами: `list calculate.c:20,29` (рисунок 12).

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

рисунок 12: просмотр определённых строк не основного файла

- Установим точку останова в файле calculate.c на строке номер 21: `break 21` (рисунок 13).

```
(gdb) break 21
Breakpoint 1 at 0x400810: file calculate.c, line 21.
(gdb)
```

рисунок 13: установка точки останова

Видим, что точка была успешно установлена.

- Выведем информацию об имеющихся в проекте точка останова. Для этого введем команду `info breakpoints` (рисунок 14).

```
(gdb) info breakpoints
Num      Type             Disp Enb Address              What
1        breakpoint       keep y  0x0000000000400810 in Calculate
                                at calculate.c:21
(gdb)
```

рисунок 14: информация об имеющихся в проекте точка останова

Можем наблюдать информацию о точке, которую мы только что установили: ее номер, тип, адрес, место установки.

- Убираем точки останова. Сначала посмотрим информацию о текущих точках `info breakpoints`, чтобы узнать номер точки, которую мы собираемся удалить. Далее удаляем ее `delete 1`, где 1 - номер точки. Снова просматриваем `info breakpoints`, чтобы убедиться в удалении точки (рисунок 15).

```
(gdb) info breakpoints
Num      Type             Disp Enb Address              What
1        breakpoint       keep y  0x0000000000400810 in Calculate
                                at calculate.c:21

(gdb) delete 1
(gdb) info breakpoints
No breakpoints or watchpoints.
(gdb)
```

рисунок 15: удаление точки останова

Видим по последнему выводу, что точка была успешно удалена.

1. Теперь с помощью утилиты `splint` нам нужно проанализировать коды файлов `calculate.c` и `main.c`. Для этого сначала установим данную утилиту. (рисунок 16).

```
[root@dasobolevskiy dasobolevskiy]# sudo yum install splint
Загружены модули: fastestmirror, langpacks
Loading mirror speeds from cached hostfile
* base: mirrors.datahouse.ru
* epel: mirror.cherryservers.com
* extras: mirrors.datahouse.ru
* updates: mirror.yandex.ru
```

рисунок 16: установка утилиты `splint`

Анализируем коды файлов `calculate.c` (рисунок 17) и `main.c` (рисунок 18) через `splint [имя файла]`.

```
dasobolevskiy@dasobolevskiy:/home/dasobolevskiy/work/os/lab_prog - □ ×
Файл Правка Вид Поиск Терминал Справка
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:33:7: Return value type double does not match declared type float:
(HUGE_VAL)
To allow all numeric types to match, use +relaxtypes.
calculate.c:39:1: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:40:7: Return value type double does not match declared type float:
(pow(Numeral, SecondNumeral))
calculate.c:42:7: Return value type double does not match declared type float:
(sqrt(Numeral))
calculate.c:44:7: Return value type double does not match declared type float:
(sin(Numeral))
calculate.c:46:7: Return value type double does not match declared type float:
(cos(Numeral))
calculate.c:48:7: Return value type double does not match declared type float:
(tan(Numeral))
calculate.c:52:7: Return value type double does not match declared type float:
(HUGE_VAL)
Finished checking --- 15 code warnings
```

рисунок 17: анализ кода calculate.c

```
[root@dasobolevskiy lab_prog]# splint main.c
Splint 3.1.2 --- 11 Oct 2015
calculate.h:5:37: 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:12:1: 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:14:12: Format argument 1 to scanf ("%s) expects char * gets char [4] *:
&Operation
Type of parameter is not consistent with corresponding code in format string.
(Use -formattype to inhibit warning)
main.c:14:9: Corresponding format code
main.c:14:1: Return value (type int) ignored: scanf("%s", &ope...
Finished checking --- 4 code warnings
[root@dasobolevskiy lab_prog]#
```

рисунок 18: анализ кода main.c

Видим, что утилита `splint` анализирует программный код, проверяет корректность задания аргументов использованных в программе функций и типов возвращаемых значений, а также обнаруживает синтаксические и семантические ошибки.

Вывод:

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

Библиография:

- [1]: [Опции компиляторов](#)
- [2]: [УСТАНОВКА ПАКЕТОВ В CENTOS 7](#)
- [3]: [Лабораторная работа №14](#)