

Отчет по выполнению лабораторной работы №13

Дисциплина: операционные системы

Астраханцева А. А.

Содержание

1	Цель работы	4
2	Задание	5
3	Теоретическое введение	6
4	Выполнение лабораторной работы	7
6	Выводы	26

Список иллюстраций

4.1	Создание каталога и файлов	7
4.2	Файл calculate.c	8
4.3	Файл calculate.h	10
4.4	Файл main.c	11
4.5	Компиляция программы	12
4.6	Файл Makefile	13
4.7	Make	14
4.8	Запуск нашего файла	15
4.9	Запуск и проверка программы	15
4.10	Команда list	16
4.11	Команда list с параметрами	16
4.12	Просмотр определённых строк не основного файла	16
4.13	Установка точки останова	17
4.14	Информация о точках останова	17
4.15	Запуск программы с установленной точкой останова	17
4.16	Значение переменной Numeral	18
4.17	Удаление точки останова	18
4.18	Вывод splint calculate.c	19
4.19	Вывод splint main.c	19

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. Ознакомиться с теоритиречским материалом
2. Выполнить все задания из “Последовательность выполнения лабораторной работы”
3. Ответить на контрольные вопросы

3 Теоретическое введение

Процесс разработки программного обеспечения обычно разделяется на следующие этапы: 1. планирование, включающее сбор и анализ требований к функционалу и другим характеристикам разрабатываемого приложения; 2. проектирование, включающее в себя разработку базовых алгоритмов и спецификаций, определение языка программирования; 3. непосредственная разработка приложения; 4. кодирование — по сути создание исходного текста программы (возможно в нескольких вариантах); 5. анализ разработанного кода; 6. сборка, компиляция и разработка исполняемого модуля; 7. тестирование и отладка, сохранение произведённых изменений; 8. 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.

4 Выполнение лабораторной работы

Создаем каталог и нужные файлы в нем (рис. 4.1).

```
4-linux»: Это недопустимый идентификатор
[aaastrakhantseva@aaastrakhantseva ~]$ mkdir -p ~/work/os/lab_prog
[aaastrakhantseva@aaastrakhantseva ~]$ cd ~/work/os/lab_prog
[aaastrakhantseva@aaastrakhantseva lab_prog]$ touch calculate.h, calculate.c, main.c
[aaastrakhantseva@aaastrakhantseva lab_prog]$ ls
calculate.c, calculate.h, main.c
[aaastrakhantseva@aaastrakhantseva lab_prog]$
```

Рис. 4.1: Создание каталога и файлов

В файл `calculate.c` записываем текст программы (рис. 4.2).

```
Открыть ▾ + calculate.h,
~/work/os/lab_prog

////////////////////////////////
// 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("Умножаем: ");
        scanf("%f", &SecondNumeral);
        return(Numeral * SecondNumeral);
    }
    else if(strncmp(Operation, "/", 1) == 0)
    {
        printf("Делим: ");
        scanf("%f", &SecondNumeral);
        return(Numeral / SecondNumeral);
    }
    return(0);
}
```

Рис. 4.2: Файл calculate.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);
}
else
return(Numeral / SecondNumeral);
}
else if(strncmp(Operation, "pow", 3) == 0)
{

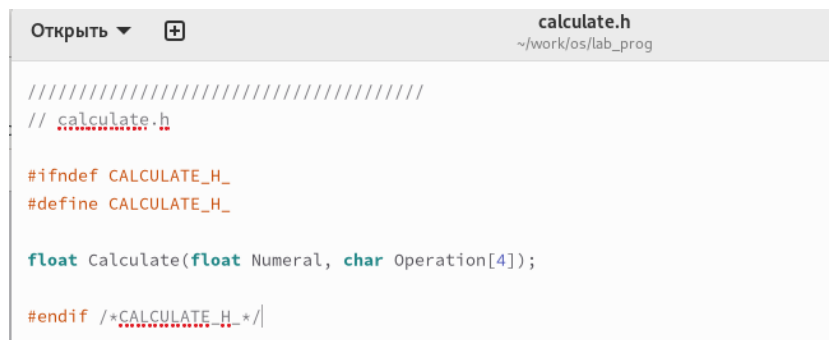
```

```

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);
}
}

```

В файл calculate.h записываем текст программы (рис. 4.3).



```

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

#ifndef CALCULATE_H_
#define CALCULATE_H_

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

#endif /*CALCULATE_H_*/

```

Рис. 4.3: Файл calculate.h

```

/////////////////////////////////////////////////

```

```
// calculate.h

#ifndef CALCULATE_H_
#define CALCULATE_H_

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

#endif /*CALCULATE_H_*/
```

В файл main.c записываем текст программы (рис. 4.4).



```

Открыть ▾ + main.c
~/work/os/lab_prog

////////////////////////////////////
// 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. 4.4: Файл main.c

```
////////////////////////////////////
// main.c

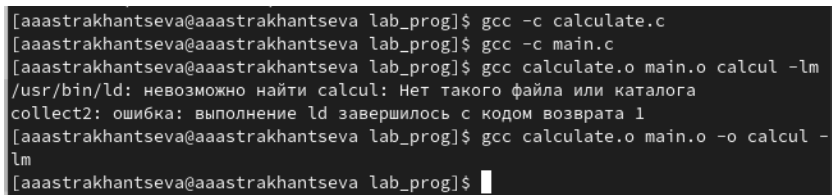
#include <stdio.h>
#include "calculate.h"
int
```

```

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;
}

```

Выполним компиляцию программы посредством gcc(рис. 4.5).



```

[aaastrakhantseva@aaastrakhantseva lab_prog]$ gcc -c calculate.c
[aaastrakhantseva@aaastrakhantseva lab_prog]$ gcc -c main.c
[aaastrakhantseva@aaastrakhantseva lab_prog]$ gcc calculate.o main.o calcul -lm
/usr/bin/ld: невозможно найти calcul: Нет такого файла или каталога
collect2: ошибка: выполнение ld завершилось с кодом возврата 1
[aaastrakhantseva@aaastrakhantseva lab_prog]$ gcc calculate.o main.o -o calcul -lm
[aaastrakhantseva@aaastrakhantseva lab_prog]$

```

Рис. 4.5: Компиляция программы

Создаем Makefile и записываем в него текст (рис. 4.6).



```
#
# Makefile
#

CC = gcc
CFLAGS = -g
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. 4.6: Файл Makefile

CC = gcc # задаем 3 переменные

CFLAGS = -g

LIBS = -lm

#Создаем файл calcul из файлов calculate.o main.o

calcul: calculate.o main.o # цель - calcul, зависимость - calculate.o main.o
\$(CC) calculate.o main.o -o calcul \$(LIBS) # команды

calculate.o: calculate.c calculate.h
\$(CC) -c calculate.c \$(CFLAGS)

```
main.o: main.c calculate.h #gcc -c main.c -g
$(CC) -c main.c $(CFLAGS)
```

```
clean: #при вызове make clean будем удалять все файлы с разрешением .o
-rm calcul *.o *~
```

Далее использую make для отладки (рис. 4.7).

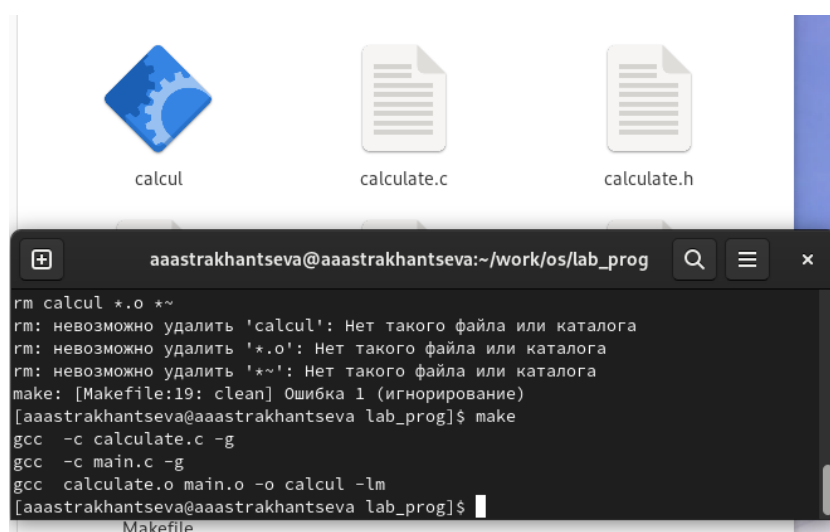


Рис. 4.7: Make

После этого запускаем gdb и вводим run, для того, чтобы запустить нашу программу (рис. 4.8).

```

make: «calcul» не такой объект: «no such file or directory».
[aaastrakhantseva@aaastrakhantseva lab_prog]$ gdb ./calcul
GNU gdb (GDB) Fedora Linux 13.1-1.fc37
Copyright (C) 2023 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".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
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...

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.fedoraproject.org/>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
Downloading separate debug info for /home/aaastrakhantseva/work/os/lab_prog/calcul
--Type <RET> for more, q to quit, c to continue without paging--
(No debugging symbols found in ./calcul)
(gdb) run

```

Рис. 4.8: Запуск нашего файла

Проверяем, что все работает корректно. 5 в третьей степени действительно равно 125.(рис. 4.9).

```

[Inferior 1 (process 9009) exited normally]
(gdb)

```

Рис. 4.9: Запуск и проверка программы

Для постраничного (по 9 строк) просмотра исходного код использую команду list: (рис. 4.10).

```
(gdb) list
1  //////////////////////////////////////////////////
2  // main.c
3
4  #include <stdio.h>
5  #include "calculate.h"
6  int
7  main (void)
8  {
9  float Numeral;
10 char Operation[4];
(gdb)
```

Рис. 4.10: Команда list

Для просмотра строк с 12 по 15 основного файла использую list с параметрами: (рис. 4.11).

```
(gdb) list 12,15
12 printf("Число: ");
13 scanf("%f",&Numeral);
14 printf("Операция (+,-,*,/,pow,sqrt,sin,cos,tan): ");
15 scanf("%s",&Operation);
(gdb)
```

Рис. 4.11: Команда list с параметрами

Для просмотра определённых строк не основного файла используйте list с параметрами: (рис. 4.12).

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

Рис. 4.12: Просмотр определённых строк не основного файла

Устанавливаю точку останова в файле calculate.c на строке номер 21: (рис. 4.13).

```
(gdb) list calculate.c:20,27
20     printf("Вычитаемое: ");
21     scanf("%f",&SecondNumeral);
22     return(Numeral - SecondNumeral);
23 }
24 else if(strncmp(Operation, "*", 1) == 0)
25 {
26     printf("Множитель: ");
27     scanf("%f",&SecondNumeral);
(gdb) break 21
Breakpoint 1 at 0x40121e: file calculate.c, line 21.
(gdb)
```

Рис. 4.13: Установка точки останова

Выведу информацию об имеющихся в проекте точка останова (рис. 4.14).

```
(gdb) info breakpoints
Num   Type      Disp Enb Address          What
1     breakpoint keep y   0x000000000040121e in Calculate at calculate.c:21
(gdb)
```

Рис. 4.14: Информация о точках останова

Запускаем программу внутри отладчика и убеждаемся, что программа остановится в момент прохождения точки останова: (рис. 4.15).

```
(gdb) run
Starting program: /home/aastrakhantseva/work/os/lab_prog/calcul

This GDB supports auto-downloading debuginfo from the following URLs:
<https://debuginfod.fedoraproject.org/>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Число: 5
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): -

Breakpoint 1, Calculate (Numeral=5, Operation=0x7fffffffdee4 "-") at calculate.c:21
21     scanf("%f",&SecondNumeral);
(gdb)
```

Рис. 4.15: Запуск программы с установленной точкой останова

Посмотрим, чему равно на этом этапе значение переменной Numeral и сравним с результатом вывода на экран после использования команды `display Numeral` (рис. 4.16).

```
21 scanf("%i",&secondNumeral);  
(gdb) print Numeral  
$1 = 5  
(gdb) display Numeral  
1: Numeral = 5  
(gdb)
```

Рис. 4.16: Значение переменной Numeral

Уберем точки останова (рис. 4.17).

```
(gdb) info breakpoints  
Num   Type             Disp Enb Address          What  
1      breakpoint       keep y   0x000000000040121e in Calculate at calculate.c:21  
       breakpoint already hit 1 time  
(gdb) delete 1  
(gdb) info breakpoints  
No breakpoints or watchpoints.  
(gdb)
```

Рис. 4.17: Удаление точки останова

С помощью утилиты `splint` попробуем проанализировать коды файлов `calculate.c` и `main.c`. Мы получили несколько предупреждений и советов по поводу того, как можно исправить текст программы (рис. 4.18 - 4.19).

```
[aaastrakhantseva@aaastrakhantseva lab_prog]$ splint calculate.c
Splint 3.1.2 --- 23 Jul 2022

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:9:31: Function parameter Operation declared as manifest array (size
        constant is meaningless)
calculate.c: (in function Calculate)
calculate.c:15:1: 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:21:1: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:27:1: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:33:1: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:34:4: 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:37:7: Return value type double does not match declared type float:
        (HUGE_VAL)
```

Рис. 4.18: Вывод splint calculate.c

```
Finished checking --- 4 code warnings
[aaastrakhantseva@aaastrakhantseva lab_prog]$ splint main.c
Splint 3.1.2 --- 23 Jul 2022

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:13:1: 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:15:12: 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:15:9: Corresponding format code
main.c:15:1: Return value (type int) ignored: scanf("%s", &Op...

Finished checking --- 4 code warnings
[aaastrakhantseva@aaastrakhantseva lab_prog]$
```

Рис. 4.19: Вывод splint main.c

Контрольные вопросы

1. Как получить информацию о возможностях программ gcc, make, gdb и др.?

С помощью man или info

2. Назовите и дайте краткую характеристику основным этапам разработки приложений в UNIX.

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.

3. Что такое суффикс в контексте языка программирования? Приведите примеры использования.

Суффикс это составная часть имени файла. Система сборки каких-либо программ (например язык java) требует, чтобы имена файлов исходного кода заканчивались на .java. 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. Каково основное назначение компилятора языка C в UNIX?

Основное назначение компилятора с языка Си заключается в компиляции всей программы в целом и получении исполняемого модуля.

5. Для чего предназначена утилита `make`?

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. 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.

6. Приведите пример структуры `Makefile`. Дайте характеристику основным элементам этого файла.

5

```
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
```

В этом примере в начале файла заданы три переменные: LIBS, 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. Назовите основное свойство, присущее всем программам отладки. Что необходимо сделать, чтобы его можно было использовать?

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. Назовите и дайте основную характеристику основным командам отладчика gdb.

`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. Опишите по шагам схему отладки программы, которую Вы использовали при выполнении лабораторной работы.

Выполнили компиляцию программы
Увидели ошибки в программе
Открыли редактор и исправили программу
Загрузили программу в отладчик gdb run — отладчик выполнил программу, мы ввели требуемые значения. программа завершилась, gdb не видит ошибок.

10. Прокомментируйте реакцию компилятора на синтаксические ошибки в программе при его первом запуске.

Отладчику не понравился формат %s для &Operation, т.к %s — символьный формат, а значит необходим только Operation.

11. Назовите основные средства, повышающие понимание исходного кода программы.

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:

cscore - исследование функций, содержащихся в программе;
splint — критическая проверка программ, написанных на языке Си.

12. Каковы основные задачи, решаемые программой splint?

Проверка корректности задания аргументов всех исполняемых функций , а также типов возвращаемых ими значений;

Поиск фрагментов исходного текста, корректных с точки зрения синтаксиса языка Си, но малоэффективных с точки зрения их реализации или содержащих в себе семантические ошибки;

Общая оценка мобильности пользовательской программы.

6 Выводы

В ходе выполнения лабораторной работы №13 я приобрела простейшие навыки разработки, анализа, тестирования и отладки приложений в ОС типа UNIX/Linux на примере создания на языке программирования С калькулятора с простейшими функциями.