

Отчёт по лабораторной работе 13

**Средства, применяемые при разработке программного обеспечения в
ОС типа UNIX/Linux**

Сидорова Наталья Андреевна

Содержание

| | | |
|----------|---------------------------------------|-----------|
| 1 | Цель работы | 5 |
| 2 | Задание | 6 |
| 3 | Теоретическое введение | 8 |
| 4 | Выполнение лабораторной работы | 10 |
| 5 | Выводы | 21 |
| | Список литературы | 22 |

Список иллюстраций

| | | |
|-----|----------------------------------|----|
| 4.1 | 1 часть кода | 11 |
| 4.2 | 2 часть кода | 12 |
| 4.3 | Код вызова функции | 12 |
| 4.4 | Код интерфейса функции | 13 |
| 4.5 | Код | 14 |
| 4.6 | Отладка программы | 15 |

Список таблиц

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

В домашнем каталоге создайте подкаталог `~/work/os/lab_prog`. 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. Выполните компиляцию программы посредством `gcc`. При необходимости исправьте синтаксические ошибки. Создайте `Makefile`. 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 код используйте команду `list: 1 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 точка останова: `info breakpoints` – Запустите программу внутри отладчика и убедитесь, что программа остановится в момент прохождения точки останова. 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 С помощью утилиты `splint` попробуйте проанализировать коды файлов `calculate.c` и `main.c`.

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.

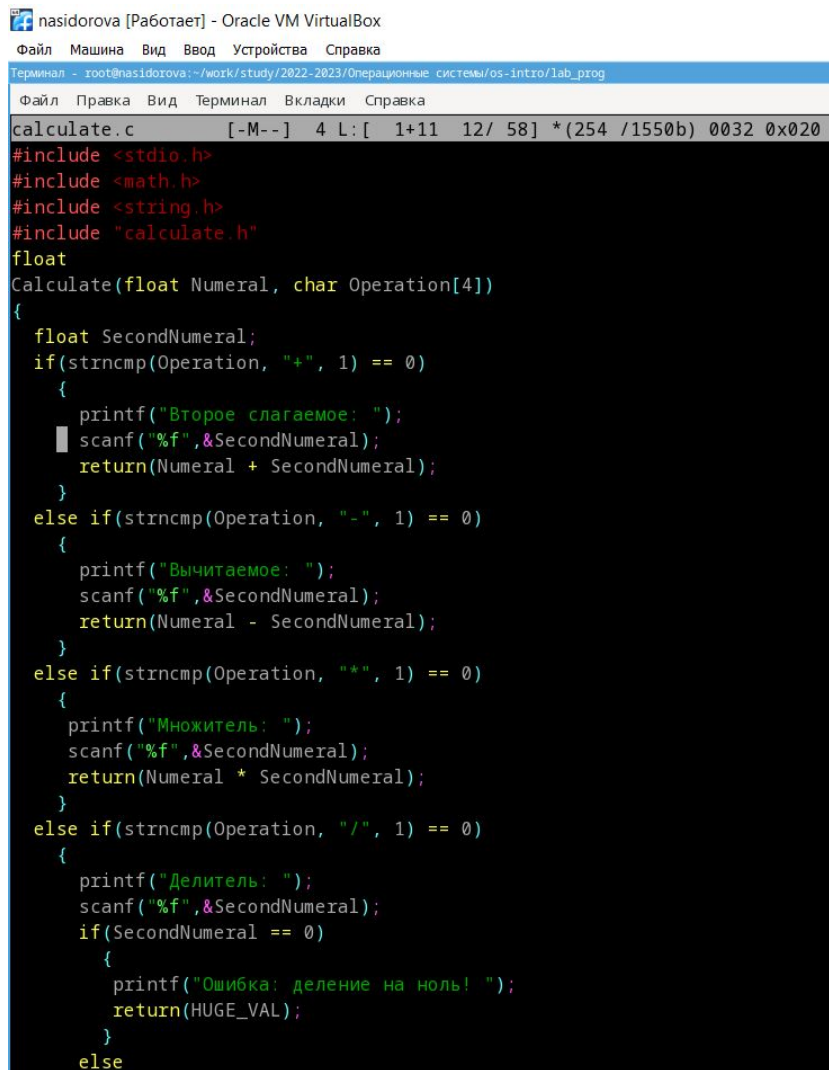
Для создания исходного текста программы разработчик может воспользоваться любым удобным для него редактором текста: vi, vim, mceditor, emacs, geany и др. После завершения написания исходного кода программы (возможно состоящей из нескольких файлов), необходимо её скомпилировать и получить исполняемый модуль.

Стандартным средством для компиляции программ в ОС типа UNIX является GCC (GNU Compiler Collection). Это набор компиляторов для разного рода языков программирования (C, C++, Java, Фортран и др.). Работа с GCC производится

при помощи одноимённой управляющей программы gсс, которая интерпретирует аргументы командной строки, определяет и осуществляет запуск нужного компилятора для входного файла. Файлы с расширением (суффиксом) .с воспринимаются gсс как программы на языке С, файлы с расширением .сс или .С — как файлы на языке С++, а файлы с расширением .о считаются объектными.

4 Выполнение лабораторной работы

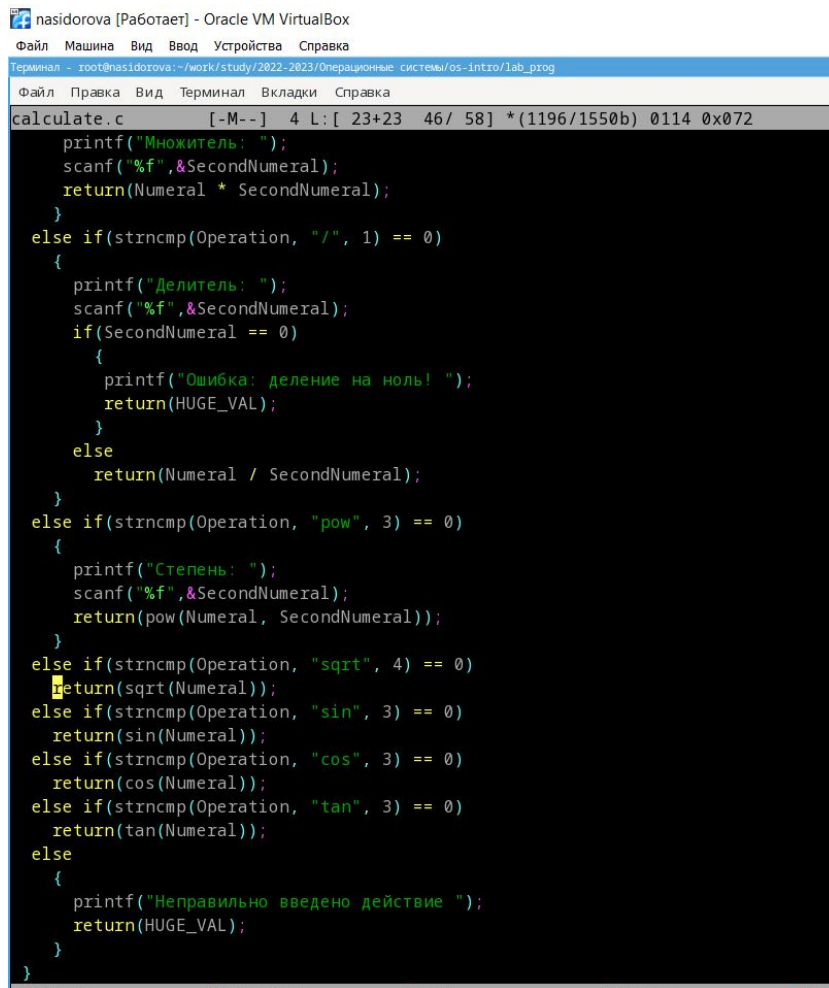
В домашнем каталоге создала подкаталог `~/work/os/lab_prog`. Создала в нём файлы: `calculate.h`, `calculate.c`, `main.c`. Код в файле `calculate.c` (рис. 4.1)



```
nasidorova [Работает] - Oracle VM VirtualBox
Файл  Машина  Вид  Ввод  Устройства  Справка
Терминал - root@nasidorova:~/work/study/2022-2023/Операционные системы/os-intro/lab_prog
Файл  Правка  Вид  Терминал  Вкладки  Справка
calculate.c  [-M--]  4 L: [ 1+11 12/ 58] *(254 /1550b) 0032 0x020
#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
```

Рис. 4.1: 1 часть кода

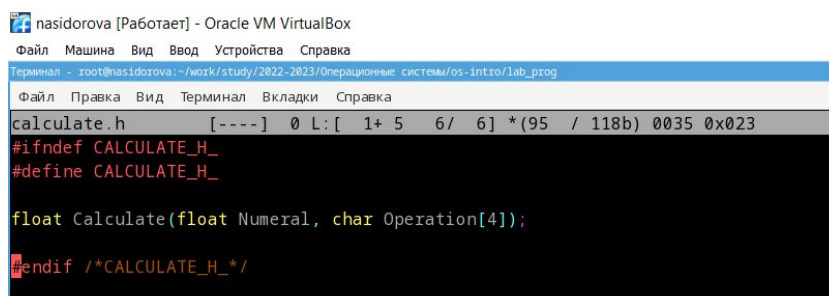
(рис. 4.2)



```
nasidorova [Работает] - Oracle VM VirtualBox
Файл Машина Вид Ввод Устройства Справка
Терминал - root@nasidorova:~/work/study/2022-2023/Операционные системы/os-intro/lab_prog
Файл Правка Вид Терминал Вкладки Справка
calculate.c [-M--] 4 L: [ 23+23 46/ 58] *(1196/1550b) 0114 0x072
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);
}
}
```

Рис. 4.2: 2 часть кода

Код в файле calculate.h (рис. 4.3)



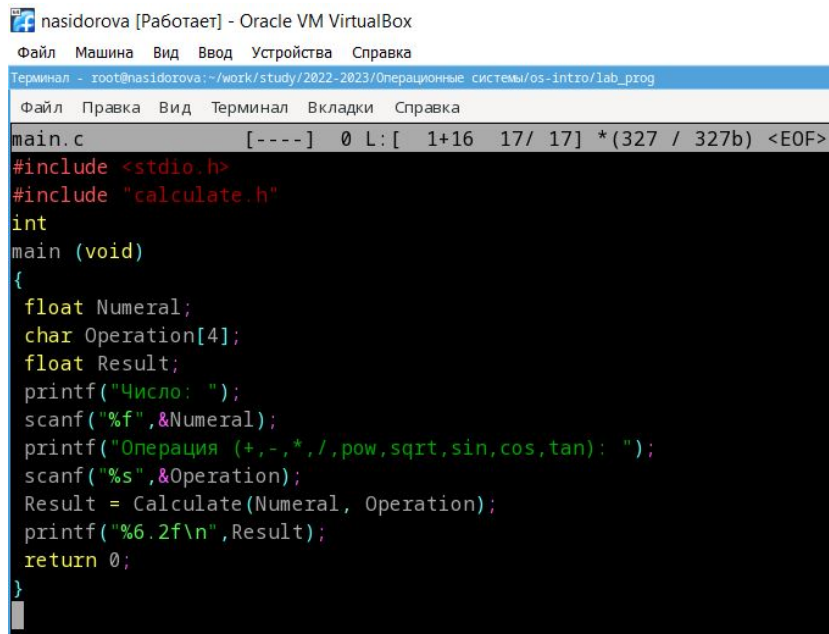
```
nasidorova [Работает] - Oracle VM VirtualBox
Файл Машина Вид Ввод Устройства Справка
Терминал - root@nasidorova:~/work/study/2022-2023/Операционные системы/os-intro/lab_prog
Файл Правка Вид Терминал Вкладки Справка
calculate.h [----] 0 L: [ 1+ 5 6/ 6] *(95 / 118b) 0035 0x023
#ifndef CALCULATE_H_
#define CALCULATE_H_

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

#endif /*CALCULATE_H_*/
```

Рис. 4.3: Код вызова функции

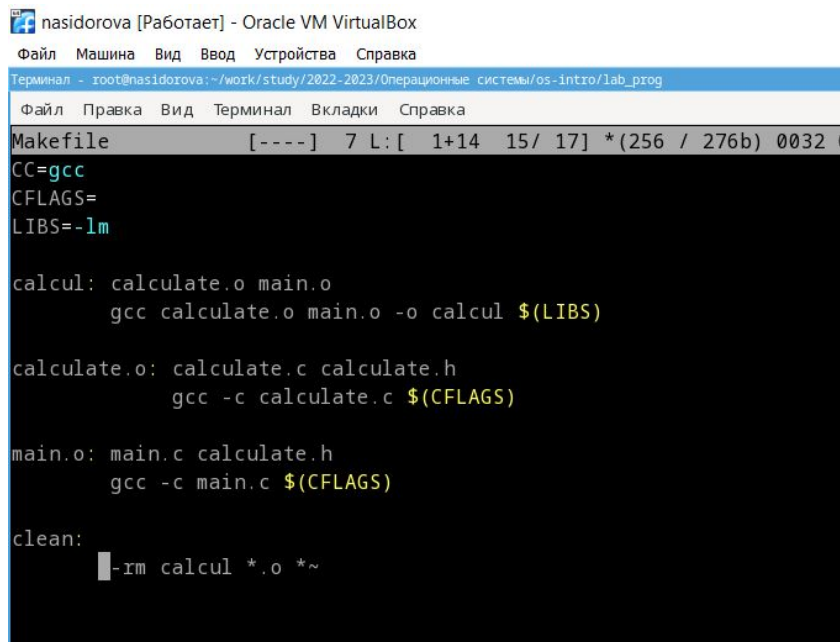
Код в файле main.c. (рис. 4.4)



```
nasidorova [Работает] - Oracle VM VirtualBox
Файл  Машина  Вид  Ввод  Устройства  Справка
Терминал - root@nasidorova:~/work/study/2022-2023/Операционные системы/os-intro/lab_prog
Файл  Правка  Вид  Терминал  Вкладки  Справка
main.c      [----]  0  L: [  1+16  17/ 17]  *(327 / 327b) <EOF>
#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: Код интерфейса функции

Выполнила компиляцию программы посредством gcc. Создала Makefile (рис. 4.5)



```
nasidorova [Работает] - Oracle VM VirtualBox
Файл  Машина  Вид  Ввод  Устройства  Справка
Терминал - root@nasidorova:~/work/study/2022-2023/Операционные системы/os-intro/lab_prog
Файл  Правка  Вид  Терминал  Вкладки  Справка
Makefile [----] 7 L: [ 1+14 15/ 17] *(256 / 276b) 0032 (
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 *~
```

Рис. 4.5: Код

Запустила отладчик GDB. Запустила программу внутри него и попробовала ввести пример, вышел нужный результат. Просмотрела строки файлов. Установила точку установки, проверила что она работает и удалила ее. С помощью утилиты splint проанализировала коды файлов calculate.c и main.c (рис. 4.6)

```

(gdb) break 21
Breakpoint 1 at 0x401247: file calculate.c, line 21.
(gdb) run
Starting program: /root/work/study/2022-2023/Операционные системы/os-intro/lab_prog/calcul
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Число: 5
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): -
Вычитаемое: 2
3.00
[Inferior 1 (process 15051) exited normally]
(gdb) run
Starting program: /root/work/study/2022-2023/Операционные системы/os-intro/lab_prog/calcul
[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=0x7fffffff124 "***") at calculate.c:21
21     else if (strcmp Operation, "***") == 0)
(gdb) print Numeral
$1 = 5
(gdb) display Numeral
1: Numeral = 5
(gdb) info breakpoints
Num      Type             Disp Enb Address                  What
1        breakpoint       keep y   0x0000000000401247 in Calculate at calculate.c:21
          breakpoint already hit 1 time
(gdb) delete 1
(gdb) info breakpoints
No breakpoints or watchpoints.
(gdb)

```

Рис. 4.6: Отладка программы

Контрольные вопросы: Как получить информацию о возможностях программ gcc, make, gdb и др.? Дополнительную информацию о этих программах можно получить с помощью функций info и man.

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

Что такое суффикс в контексте языка программирования? Приведите примеры использования. Использование суффикса “.с” 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 .с компилятор распознает, что файл abcd.c должен компилироваться, а по суффиксу .о, 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 bzipdiff -p1 выводит префиксы в форме которая подходит для команды patch -p1.

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.

Для чего предназначена утилита make? 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.

Приведите пример структуры Makefile. Дайте характеристику основным элементам этого файла. makefile для программы abcd.c мог бы иметь вид:

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

```

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.

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.

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;

Опишите по шагам схему отладки программы, которую Вы использовали при выполнении лабораторной работы. Выполнили компиляцию программы 2) Увидели ошибки в программе Открыли редактор и исправили программу Загрузили программу в отладчик `gdb run` — отладчик выполнил программу, мы ввели требуемые значения. программа завершена, `gdb` не видит ошибок.

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`.

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:

cscope - исследование функций, содержащихся в программе;

splint — критическая проверка программ, написанных на языке Си.

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.

5 Выводы

В процессе выполнения лабораторной работы 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.

Список литературы