Отчет по лабораторной работе №14

Дисциплина

Филиппова Анна Дмитриевна

Содержание

1 Цель работы **4**

2 Выполнение лабораторной работы **5**

3 Выводы **15**

4 Контрольные вопросы **16**

5 Библиография **21**

Список иллюстраций

2.1 Создаем подкаталог 5
2.2 Создаем файлы 5
2.3 Пишем командные файлы 6
2.4 Пишем командные файлы 6
2.5 Пишем командные файлы 6
2.6 Пишем командные файлы 7
2.7 Пишем командные файлы 7
2.8 Пишем командные файлы 7
2.9 Компиляция программы 8
2.10 Создание файла 8
2.11 Пишем Makefile 9
2.12 Исправляем Makefile 9
2.13 Компиляция файлов 10
2.14 Запуск отладчика 10
2.15 run 10
2.16 list 11
2.17 list 12,15 11
2.18 list calculate.c:20,29 11
2.19 Точка останова 12
2.20 Информация о точках останова 12
2.21 Проверка точки останова 12
2.22 Информация о точках останова 13
2.23 Информация о точках останова 13
2.24 Убираем точку оастанова 13
2.25 Установка splint 14
2.26 Splint 14
2.27 Splint 14

1 Цель работы

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

2 Выполнение лабораторной работы

1. В домашнем каталоге создаем подкаталог ~/work/os/lab_prog с помощью команды «mkdir -p ~/work/os/lab_prog». (рис. -fig. 2.1)

Рис. 2.1: Создаем подкаталог

2. Создаем в каталоге файлы: calculate.h, calculate.c, main.c, используя команды «cd ~/work/os/lab_prog» и «touch 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. (рис. -fig. 2.2)

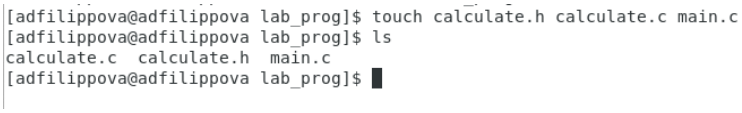

Рис. 2.2: Создаем файлы

Открыв редактор Emacs, приступаем к редактированию созданных файлов. Реализация функций калькулятора в файле calculate.c (рис. -fig. 2.3) (рис. -fig. 2.4) (рис. -fig. 2.5) (рис. -fig. 2.6)

```
////////////////////////////////////
// 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);
U:---  calculate.c    Top L20    (C/l Abbrev)
```

Рис. 2.3: Пишем командные файлы

```
    {
      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
U:---  calculate.c    28% L20    (C/l Abbrev)
```

Рис. 2.4: Пишем командные файлы

```
          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
U:---  calculate.c    56% L42    (C/l Abbrev)
```

Рис. 2.5: Пишем командные файлы

```
  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. 2.6: Пишем командные файлы

Интерфейсный файл calculate.h, описывающий формат вызова функции калькулятора. (рис. -fig. 2.7)

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

#ifndef CALCULATE_H_
#define CALCULATE_H_

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

#endif /*CALCULATE_H_*/
```

Рис. 2.7: Пишем командные файлы

Основной файл main.c, реализующий интерфейс пользователя к калькулятору. (рис. -fig. 2.8)

```
////////////////////////////////////
// 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;
}
U:---  main.c         All L19    (C/l Abbrev)
```

Рис. 2.8: Пишем командные файлы

3. Выполняем компиляцию программы посредством gcc, используя команды «gcc -c calculate.c», «gcc -c main.c» и «gcc calculate.o main.o -o calcul -lm». (рис. -fig. 2.9)

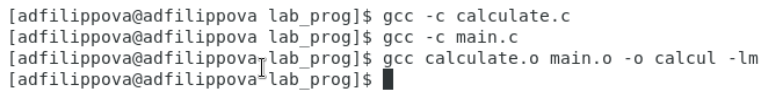

Рис. 2.9: Компиляция программы

4. Ошибок не возникло.

5. Создаем Makefile с нужным содержанием. Данный файл необходим для автоматической компиляции файлов calculate.c (цель calculate.o), main.c (цель main.o), а также их объединения в один исполняемый файл calcul (цель calcul). Цель clean нужна для автоматического удаления файлов. Переменная CC отвечает за утилиту для компиляции. Переменная CFLAGS отвечает за опции в данной утилите. Переменная LIBS отвечает за опции для объединения объектных файлов в один исполняемый файл. (рис. -fig. 2.10) (рис. -fig. 2.11)

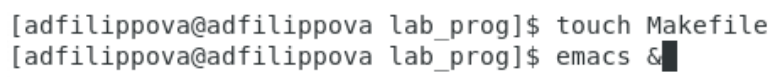

Рис. 2.10: Создание файла

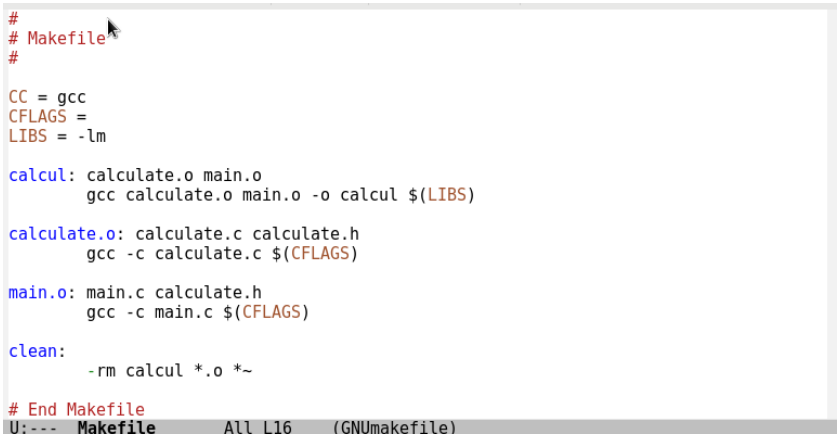

Рис. 2.11: Пишем Makefile

6. Исправляем Makefile перед использованием gdb. В переменную CFLAGS добавляем опцию -g, необходимую для компиляции объектных файлов и их использования в программе отладчика GDB. Делаем так, что утилита компиляции выбирается с помощью переменной CC. (рис. -fig. 2.12)

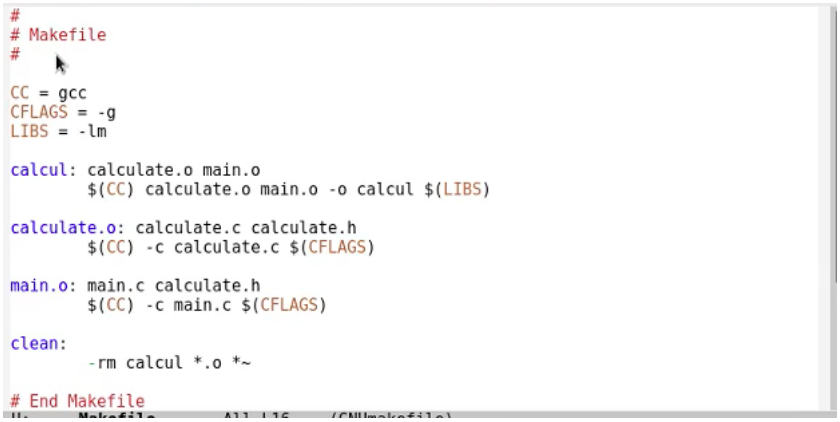

Рис. 2.12: Исправляем Makefile

После этого удаляем исполняемые и объектные файлы из каталога с помощью команды «make clear». Выполняем компиляцию файлов, используя команды «make calculate.o», «make main.o», «male calcul». (рис. -fig. 2.13)

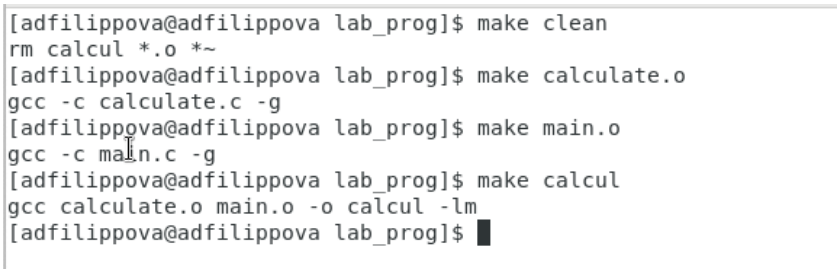

Рис. 2.13: Компиляция файлов

Далее с помощью gdb выполняем отладку программы calcul. Запускаем отладчик GDB, загрузив в него программу для отладки, используя команду: «gdb ./calcul». (рис. -fig. 2.14)

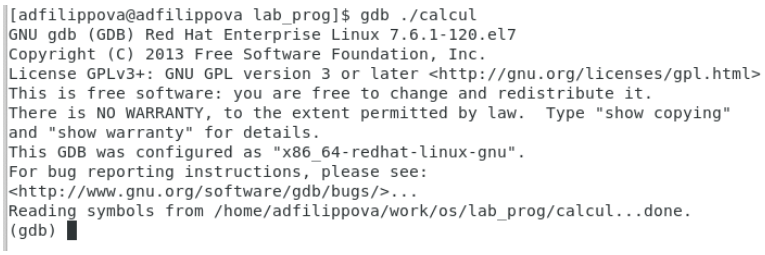

Рис. 2.14: Запуск отладчика

Для запуска программы внутри отладчика вводим команду «run». (рис. -fig. 2.15)

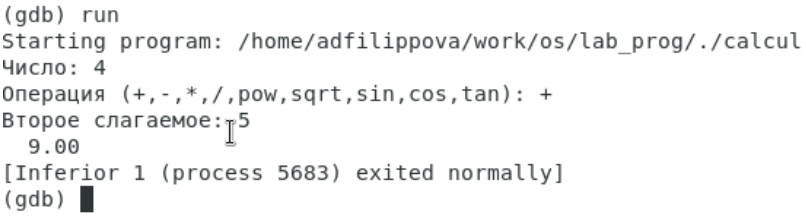

Рис. 2.15: run

Для постраничного (по 9 строк) просмотра исходного кода используем команду «list». (рис. -fig. 2.16)

```
(gdb) list
4       #include <stdio.h>
5       #include "calculate.h"
6
7       int
8       main (void)
9       {
10        float Numeral;
11        char Operation[4];
12        float Result;
13        printf("Число: ");
(gdb)
```

Рис. 2.16: list

Для просмотра строк с 12 по 15 основного файла используем команду «list 12,15». (рис. -fig. 2.17)

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

Рис. 2.17: list 12,15

Для просмотра определённых строк не основного файла используем команду «list calculate.c:20,29». (рис. -fig. 2.18)

```
(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          }
```

Рис. 2.18: list calculate.c:20,29

Устанавливаем точку останова в файле calculate.c на строке номер 21,используя команды «list calculate.c:20,27» и «break 21». (рис. -fig. 2.19)

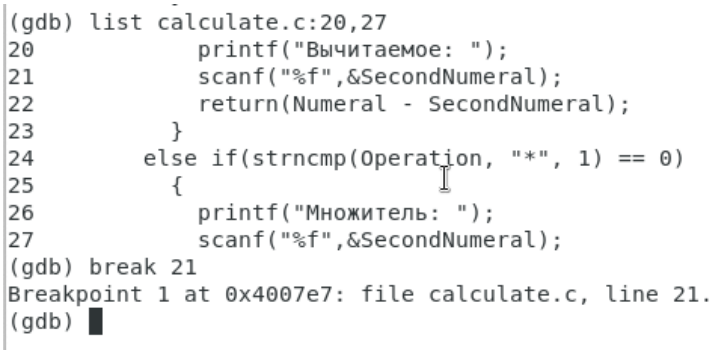

Рис. 2.19: Точка останова

Выводим информацию об имеющихся в проекте точках останова с помощью команды «info breakpoints». (рис. -fig. 2.20)

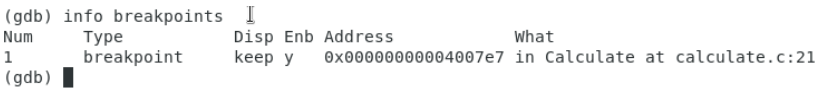

Рис. 2.20: Информация о точках останова

Запускаем программу внутри отладчика и убедилась, что программа остановилась в момент прохождения точки останова. Использовала команды «run», «5», «−» и «backtrace». (рис. -fig. 2.21)

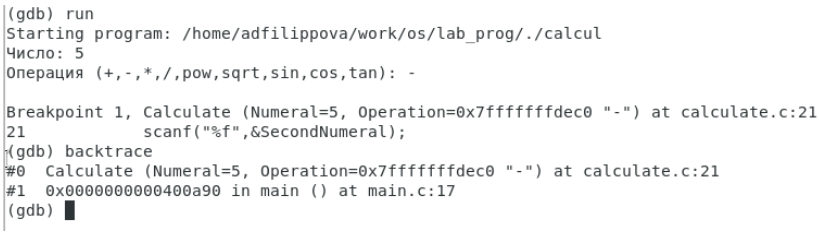

Рис. 2.21: Проверка точки останова

Посматриваем, чему равно на этом этапе значение переменной Numeral,введя команду «print Numeral». (рис. -fig. 2.22)

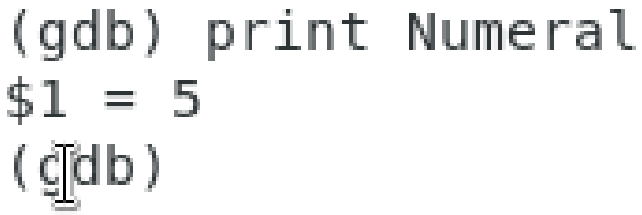

Рис. 2.22: Информация о точках останова

Сравниваем с результатом вывода на экран после использования команды «display Numeral». Значения совпадают. (рис. -fig. 2.23)

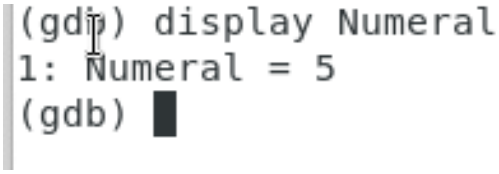

Рис. 2.23: Информация о точках останова

Убираем точки останова с помощью команд «info breakpoints» и «delete 1». (рис. -fig. 2.24)

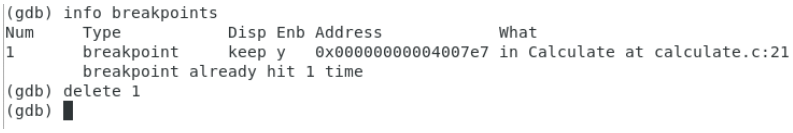

Рис. 2.24: Убираем точку оастанова

7. С помощью утилиты splint анализируем коды файлов calculate.c и main.c. Предварительно устанавливаем данную утилиту с помощью команды «yum install splint». Далее используем команду «splint calculate.c» и «splint main.c». С помощью утилиты splint выяснилось, что в файлах calculate.c и main.c присутствует функция чтения scanf, возвращающая целое число (тип int), но эти числа не используются и нигде не сохранятся. Утилита вывела предупреждение о том, что в файле calculate.c происходит сравнение вещественного числа с нулем. Также возвращаемые значения (тип double) в функциях pow, sqrt, sin, cos и tan записываются в переменную типа float, что свидетельствует о потери данных. (рис. -fig. 2.25) (рис. -fig. 2.26) (рис. -fig. 2.27)

```
[adfilippova@adfilippova lab_prog]$ su
Пароль:
[root@adfilippova lab_prog]# yum install splint
Загружены модули: fastestmirror, langpacks
Loading mirror speeds from cached hostfile
 * base: mirror.mijn.host
 * epel: ftp.nluug.nl
 * extras: ams.edge.kernel.org
 * updates: ftp.nluug.nl
Разрешение зависимостей
--> Проверка сценария
---> Пакет splint.x86_64 0:3.1.2-15.el7 помечен для установки
--> Проверка зависимостей окончена

Зависимости определены
```

Рис. 2.25: Установка splint

```
[adfilippova@adfilippova lab_prog]$ splint calculate.c
Splint 3.1.2 --- 11 Oct 2015

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:15:7: 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:21:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:27:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:33:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:34:10: 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:37:10: Return value type double does not match declared type float:
                     (HUGE VAL)
```

Рис. 2.26: Splint

```
[adfilippova@adfilippova lab_prog]$ splint main.c
Splint 3.1.2 --- 11 Oct 2015

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:14:3: 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:16:14: 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:16:11: Corresponding format code
main.c:16:3: Return value (type int) ignored: scanf("%s", &Ope...

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

Рис. 2.27: Splint

3 Выводы

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.

4 Контрольные вопросы

1. Чтобы получить информацию о возможностях программ gcc, make, gdb и др. нужно воспользоваться командой man или опцией -help (-h) для каждой команды.

2. Процесс разработки программного обеспечения обычно разделяется на следующие этапы:

- планирование, включающее сбор и анализ требований кфункционалу и другим характеристикам разрабатываемого приложения;
- проектирование, включающее в себя разработку базовых алгоритмов и спецификаций, определение языка программирования;
- непосредственная разработка приложения: о кодирование – по сути создание исходного текста программы (возможно в нескольких вариантах); – анализ разработанного кода; о сборка, компиляция и разработка исполняемого модуля; о 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. Для имени входного файла суффикс определяет какая компиляция требуется. Суффиксы указывают на тип объекта. 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. Например, в команде «gcc -c main.c»: gcc по расширению (суффиксу) .c распознает 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 -o и в качестве параметра задать имя создаваемого файла: «gcc -o hello main.c».

4. Основное назначение компилятора языка Си в UNIX заключается вкомпиляции 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 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: … : … <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 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. Общий синтаксис Makefile имеет вид: target1 [target2…]:[:] [dependment1…] [(tab)commands] [#commentary] [(tab)commands] [#commentary] Здесь знак # определяет начало коммента-

рия (содержимое от знака # и до конца строки не будет обрабатываться. Одинарное двоеточие указывает на то, что последовательность команд должна содержаться водной 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 = gcc CFLAGS = # Compile abcd.c normaly abcd: abcd.c $(CC) -o abcd $(CFLAGS) abcd.c clean: -rm abcd .*o* ~ # End Makefile for abcd.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 clean производит 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 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

8. Основные команды отладчика 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 выполнениявызываемых 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 Для выхода из gdb можно воспользоваться командой quit (или её сокращённым вариантом q) или комбинацией клавиш Ctrl-d. Более подробную информацию по работе с gdb можно получить с помощью команд gdb -h и man gdb.

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 массива символов уже является указателем на первый элемент этого 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 C анализатор splint генерирует 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 работепрограммы, переменные с некорректно заданными значениями и типами и многое другое.

5 Библиография

1. Кулябов Д.С. Операционные системы: лабораторные работы: учебное пособие / Д.С. Кулябов, М.Н. Геворкян, А.В. Королькова, А.В. Демидова. — М. : Изд-во РУДН, 2016. — 117 с. — ISBN 978-5-209-07626-1 : 139.13; То же [Электронный ресурс]. — URL: http://lib.rudn.ru/MegaPro2/Download/MObject/6118.

2. Робачевский А.М. Операционная система UNIX [текст] : Учебное пособие / А.М. Робачевский, С.А. Немнюгин, О.Л. Стесик. — 2-е изд., перераб. и доп. — СПб. : БХВ-Петербург, 2005, 2010. — 656 с. : ил. — ISBN 5-94157-538-6 : 164.56. (ЕТ 60)

3. Таненбаум Эндрю. Современные операционные системы [Текст] / Э. Таненбаум. — 2-е изд. — СПб. : Питер, 2006. — 1038 с. : ил. — (Классика Computer Science). — ISBN 5-318-00299-4 : 446.05. (ЕТ 50)