

Лабораторная работа 14

Дисциплина: Операционные системы

Куликов Максим Игоревич

Содержание

1	Цель работы	5
2	Задание	6
3	Выполнение лабораторной работы	7
4	Выводы	16
5	Контрольные вопросы	17

Список таблиц

Список иллюстраций

3.1	Создание файлов	7
3.2	Скрипты	8
3.3	Компиляция	8
3.4	Мейкфайл	9
3.5	Изменение содержимого	10
3.6	make clean	11
3.7	Конвертация	11
3.8	Тест	12
3.9	list	13
3.10	breakpoint	13
3.11	calculate.c	14
3.12	main.c	15

1 Цель работы

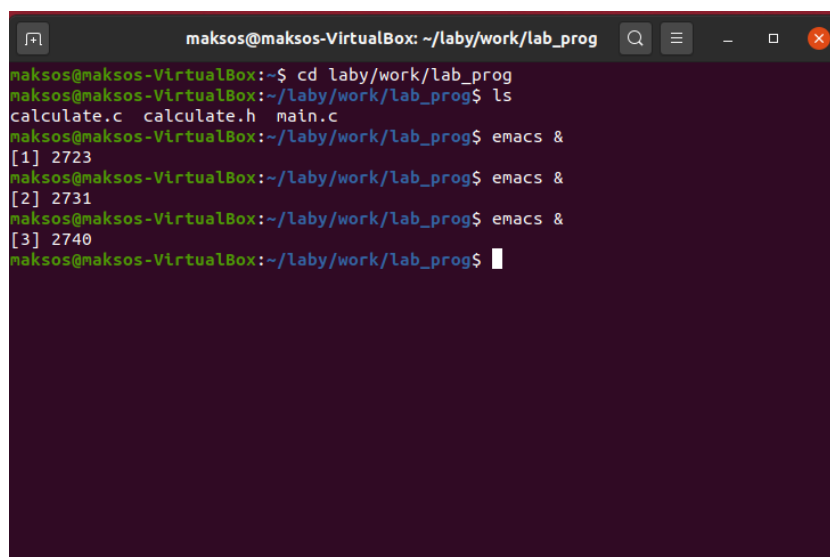
Приобрести простейшие навыки разработки, анализа, тестирования и отладки приложений в ОС типа UNIX/Linux на примере создания на языке программирования С калькулятора с простейшими функциями.

2 Задание

1. Ознакомиться с теоретическим материалом.
2. Выполнить работу.

3 Выполнение лабораторной работы

1. Создаю новый каталог и создаю 3 скрипта (рис. -fig. 3.1)

A screenshot of a terminal window titled 'maksos@maksos-VirtualBox: ~/laby/work/lab_prog'. The terminal shows the following commands and output:

```
maksos@maksos-VirtualBox:~$ cd laby/work/lab_prog
maksos@maksos-VirtualBox:~/laby/work/lab_prog$ ls
calculate.c calculate.h main.c
maksos@maksos-VirtualBox:~/laby/work/lab_prog$ emacs &
[1] 2723
maksos@maksos-VirtualBox:~/laby/work/lab_prog$ emacs &
[2] 2731
maksos@maksos-VirtualBox:~/laby/work/lab_prog$ emacs &
[3] 2740
maksos@maksos-VirtualBox:~/laby/work/lab_prog$
```

Рис. 3.1: Создание файлов

2. Три скрипта. (рис. -fig. 3.2)

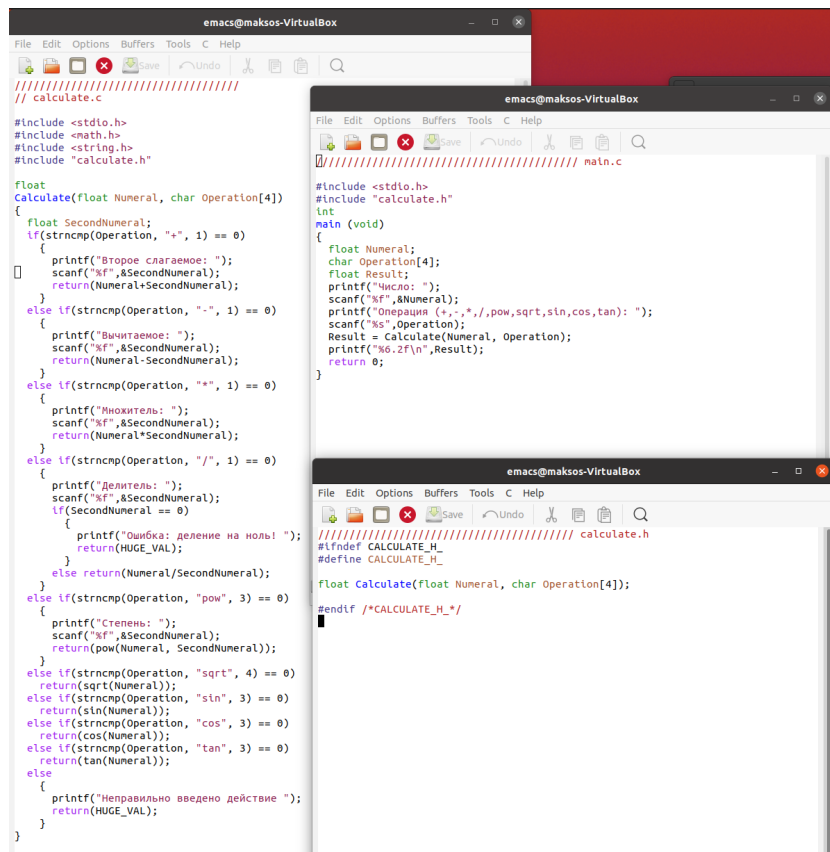


Рис. 3.2: Скрипты

3. Выполняю компиляцию файлов. (рис. -fig. 3.3)

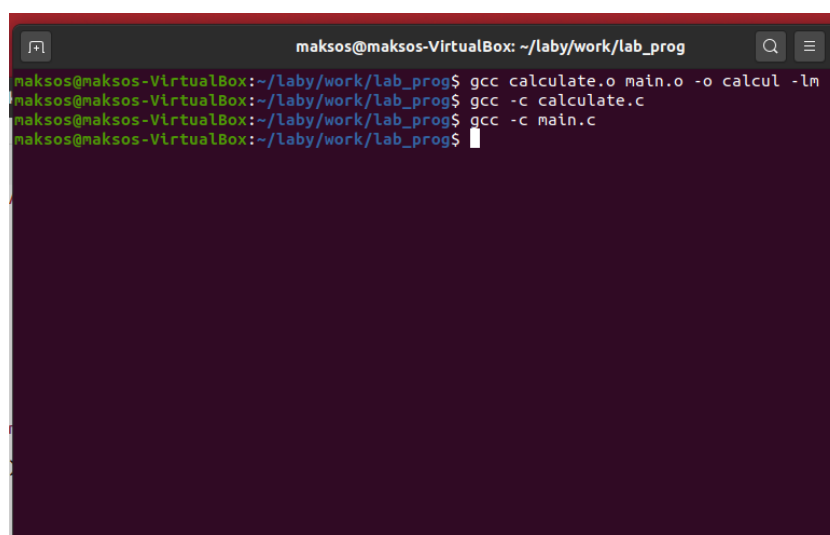
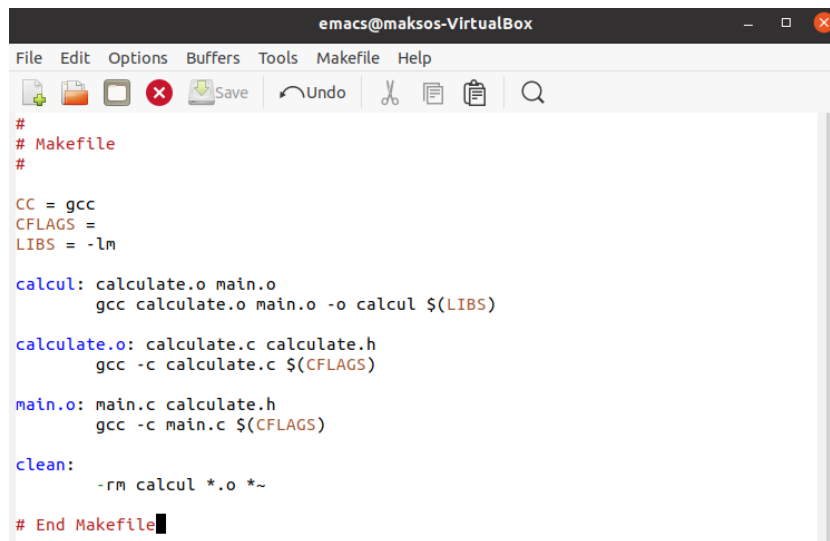


Рис. 3.3: Компиляция

5. Создание Makefile. Данный файл необходим для автоматической компиляции файлов calculate.c(цельcalculate.o), main.c(цельmain.o), а также их объединения в один исполняемый файл calcul(цельcalcul). 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. 3.4)



```
#
# 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. 3.4: Мейкфайл

6. Вношу изменения в файл. (рис. -fig. 3.5)

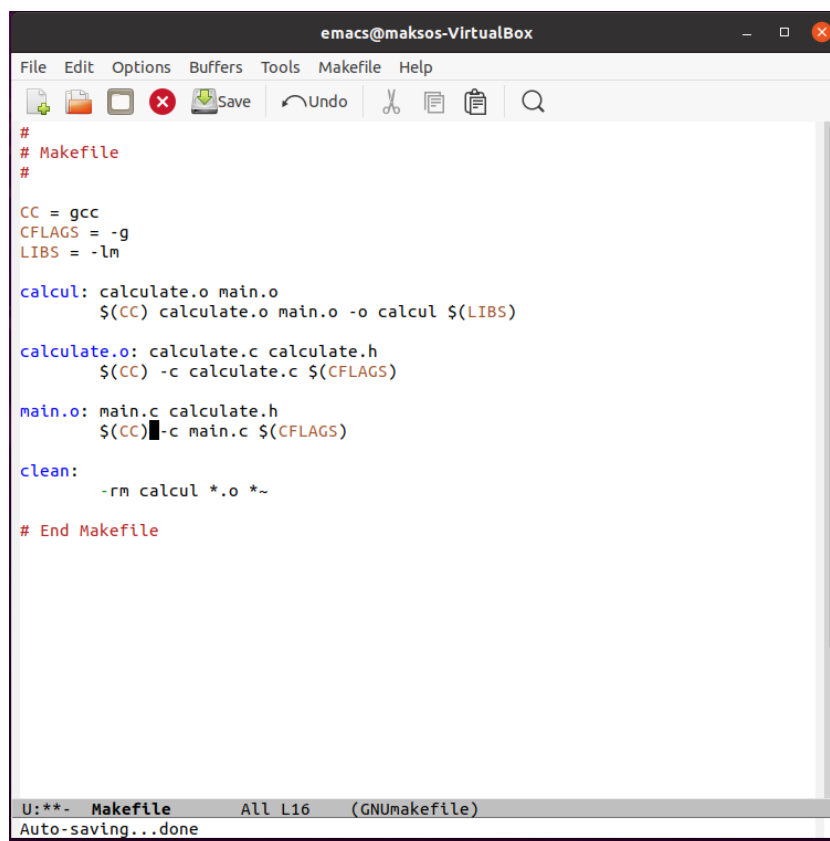
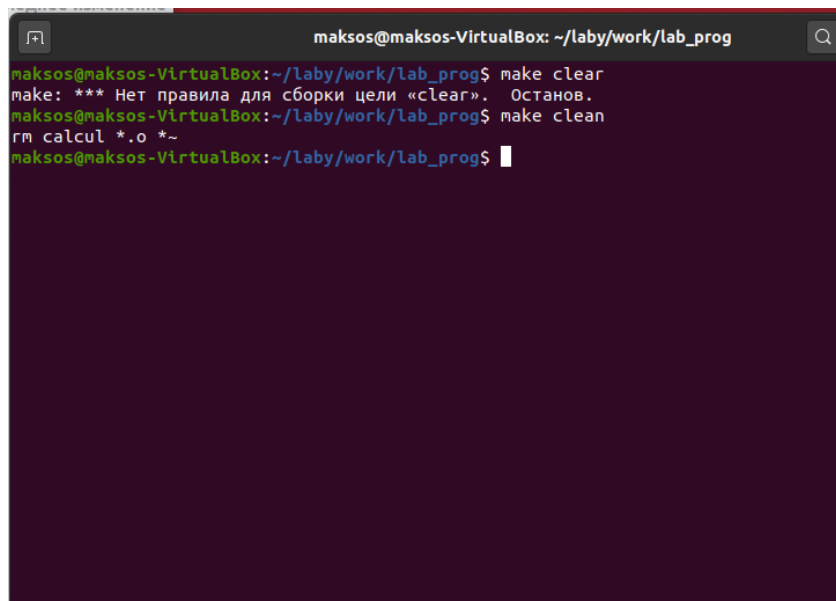


Рис. 3.5: Изменение содержимого

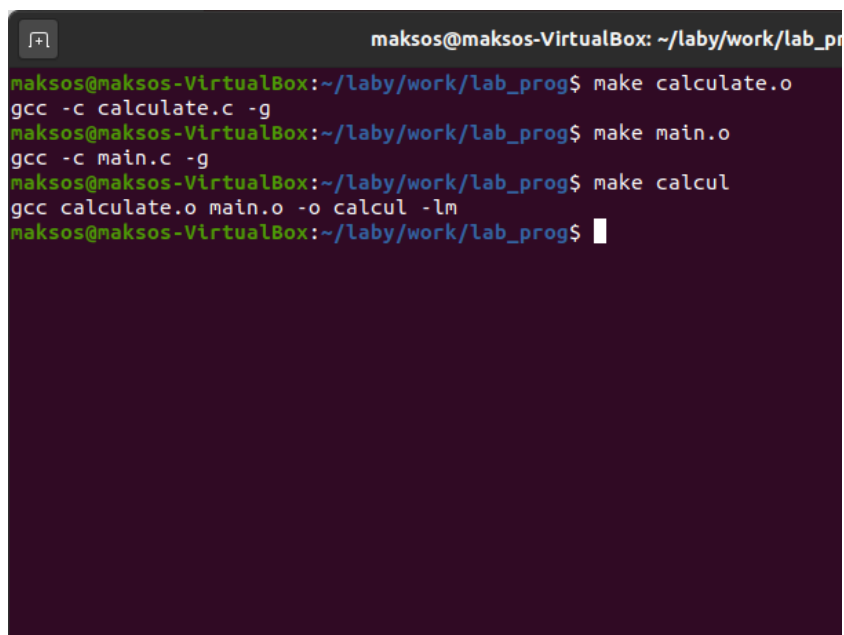
7. Выполняю make clean. Удаляю ненужные файлы. (рис. -fig. 3.6)



```
maksos@maksos-VirtualBox: ~/laby/work/lab_prog
maksos@maksos-VirtualBox:~/laby/work/lab_prog$ make clear
make: *** Нет правила для сборки цели «clear». Останов.
maksos@maksos-VirtualBox:~/laby/work/lab_prog$ make clean
rm calcul *.o *~
maksos@maksos-VirtualBox:~/laby/work/lab_prog$
```

Рис. 3.6: make clean

8. Конвертирую все файлы (рис. -fig. 3.7)



```
maksos@maksos-VirtualBox: ~/laby/work/lab_prog
maksos@maksos-VirtualBox:~/laby/work/lab_prog$ make calculate.o
gcc -c calculate.c -g
maksos@maksos-VirtualBox:~/laby/work/lab_prog$ make main.o
gcc -c main.c -g
maksos@maksos-VirtualBox:~/laby/work/lab_prog$ make calcul
gcc calculate.o main.o -o calcul -lm
maksos@maksos-VirtualBox:~/laby/work/lab_prog$
```

Рис. 3.7: Конвертация

9. Тестирую программу (рис. -fig. 3.8)

```
maksos@maksos-VirtualBox: ~/laby/work/lab_prog
gcc -c calculate.c -g
maksos@maksos-VirtualBox:~/laby/work/lab_prog$ make main.o
gcc -c main.c -g
maksos@maksos-VirtualBox:~/laby/work/lab_prog$ make calcul
gcc calculate.o main.o -o calcul -lm
maksos@maksos-VirtualBox:~/laby/work/lab_prog$ gdb ./calcul
GNU gdb (Ubuntu 9.1-0ubuntu1) 9.1
Copyright (C) 2020 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-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://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...
(gdb) run
Starting program: /home/maksos/laby/work/lab_prog/calcul
Число: 6
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): *
Множитель: 8
48.00
[Inferior 1 (process 3554) exited normally]
(gdb) █
```

Рис. 3.8: Тест

10. Командой list вывожу содержимое файлов (рис. -fig. 3.6)

```

[Inferior 1 (process 3554) exited normally]
(gdb) list
1      ////////////////////////////////////////////////// main.c
2
3      #include <stdio.h>
4      #include "calculate.h"
5      int
6      main (void)
7      {
8          float Numeral;
9          char Operation[4];
10         float Result;
(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) list calculate.c 12,18
Function "calculate.c 12" not defined.
(gdb) list calculate.c:12,18
12         float SecondNumeral;
13         if(strncmp(Operation, "+", 1) == 0)
14         {
15             printf("Второе слагаемое: ");
16             scanf("%f",&SecondNumeral);
17             return(Numeral+SecondNumeral);
18         }
(gdb) █

```

Рис. 3.9: list

11. Ставлю breakpoint. Запускаю программу и проверяю работает ли он. Работает. Удаляю его (рис. -fig. 3.10)

```

(gdb) list calculate.c:20,27
20         {
21             printf("Вычитаемое: ");
22             scanf("%f",&SecondNumeral);
23             return(Numeral-SecondNumeral);
24         }
25         else if(strncmp(Operation, "*", 1) == 0)
26         {
27             printf("Множитель: ");
(gdb) break 21
Breakpoint 1 at 0x555555552dd: file calculate.c, line 21.
(gdb) info breakpoints
Num   Type             Disp Enb Address            What
1     breakpoint       keep y   0x0000555555552dd in Calculate at calculate.c:21
(gdb) run
Starting program: /home/maksos/laby/work/lab_prog/calcul
Число: 9
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): -
Breakpoint 1, Calculate (Numeral=9, Operation=0x7fffffffdf64 "-") at calculate.c:21
21             printf("Вычитаемое: ");
(gdb) print Numeral
$1 = 9
(gdb) display Numeral
1: Numeral = 9
(gdb) info breakpoints
Num   Type             Disp Enb Address            What
1     breakpoint       keep y   0x0000555555552dd in Calculate at calculate.c:21
1     breakpoint already hit 1 time
(gdb) delete 1
(gdb) █

```

Рис. 3.10: breakpoint

13. Устанавливаю splint . Запускаю его для calculate.c и 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 сохраняются. 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 потере данных. (рис. -fig. 3.11, рис. -fig. 3.12)

```
maksos@maksos-VirtualBox:~/laby/work/lab_prog$ splint calculate.c
Splint 3.1.2 --- 20 Feb 2018

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)
calculate.c:10:31: Function parameter Operation declared as manifest array
(size constant is meaningless)
calculate.c: (in function Calculate)
calculate.c:16:7: 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:22:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:28:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:34:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:35:10: 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:38:10: Return value type double does not match declared type float:
(HUGE_VAL)
    To allow all numeric types to match, use +relaxtypes.
calculate.c:45:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:46:13: Return value type double does not match declared type float:
(pow(Numeral, SecondNumeral))
calculate.c:49:11: Return value type double does not match declared type float:
(sqrt(Numeral))
calculate.c:51:11: Return value type double does not match declared type float:
(sin(Numeral))
calculate.c:53:11: Return value type double does not match declared type float:
(cos(Numeral))
calculate.c:55:11: Return value type double does not match declared type float:
(tan(Numeral))
calculate.c:59:13: Return value type double does not match declared type float:
(HUGE_VAL)
Finished checking --- 15 code warnings
```

Рис. 3.11: calculate.c

```
maksos@maksos-VirtualBox:~/laby/work/lab_prog$ splint main.c
Splint 3.1.2 --- 20 Feb 2018

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:12:3: 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:14:3: Return value (type int) ignored: scanf("%s", Oper...
Finished checking --- 3 code warnings
```

Рис. 3.12: main.c

4 Выводы

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

5 Контрольные вопросы

1. Как получить более полную информацию о программах: gcc, make, gdb и др.? Дополнительную информацию о этих программах можно получить с помощью функций info и man.
2. 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.
3. Что такое суффиксы и префиксы? Основное их назначение. Приведите примеры их использования. 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. Основное назначение компилятора с языка Си 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`. 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`.
6. Приведите структуру `make-файла`. 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`. Второй способ позволяет включать в исполняемый модуль `testabcd` возможность выполнить процесс отладки на уровне исходного текста.

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; – slist – 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 передается 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.

- 1) Выполнили компиляцию программы 2) Увидели ошибки в программе 3) Открыли редактор и исправили программу 4) Загрузили программу в отладчик gdb 5) run — отладчик выполнил программу, мы ввели требуемые значения. 6) программа завершена, gdb не видит ошибок.
10. Прокомментируйте реакцию компилятора на синтаксические ошибки в программе при его первом запуске. 1 и 2.) Мы действительно забыли закрыть комментарии; 3.) 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: – cscope - исследование функций, содержащихся в программе; – splint — критическая проверка программ, написанных на языке Си.