Отчёт по лабораторной работе

Лабораторная работа № 14.

Lukashov Nikita

Содержание

List of Tables

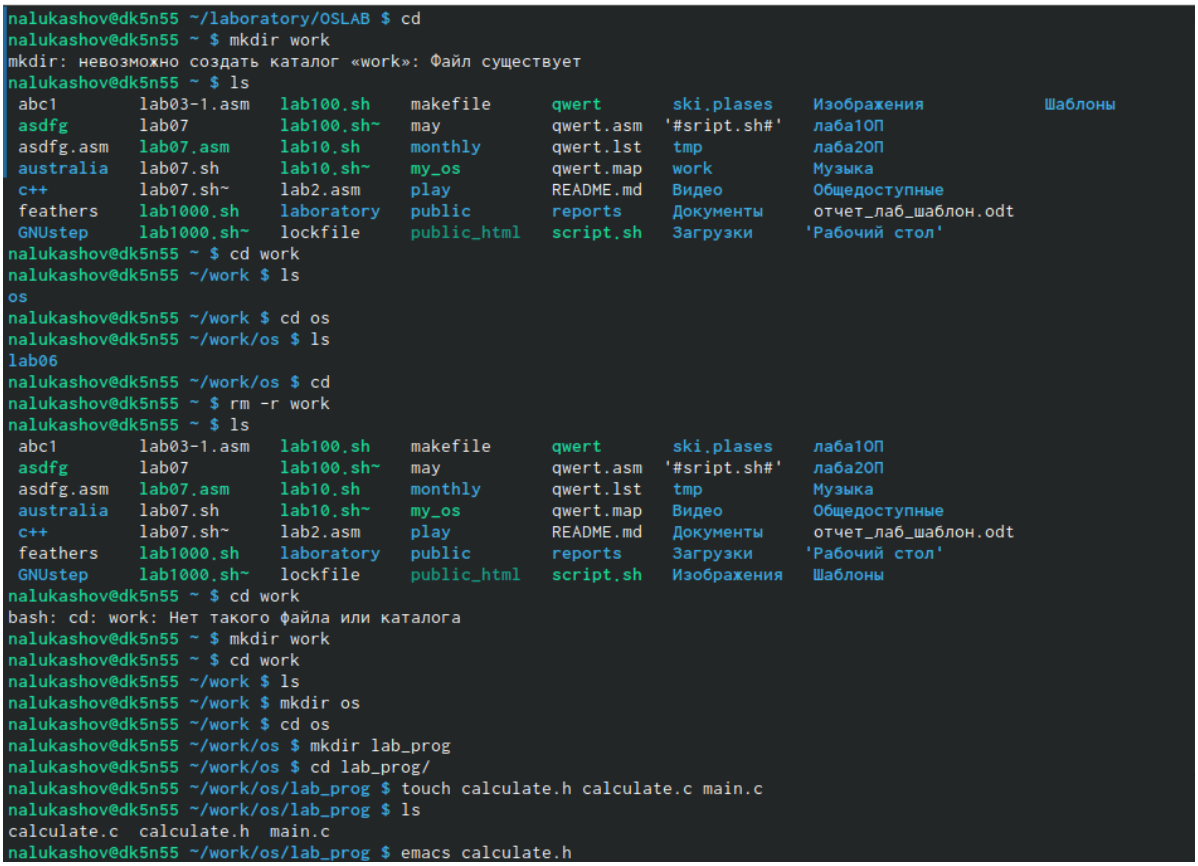
List of Figures

1 Цель работы

Изучить основы программирования в оболочке ОС UNIX. Научиться писать более сложные командные файлы с использованием логических управляющих кон-струкций и циклов.

2 Выполнение лабораторной работы

1. В домашнем каталоге создал подкаталог ~/work/os/lab_prog.



2. Создал в нём файлы: calculate.h, calculate.c, main.c. Это примитивнейший калькулятор, способный складывать, вычитать, умножать, делить, возводить число в степень, вычислять квадратный корень, вычислять sin, cos, tan. При запуске он запрашивает первое число, операцию, второе число. После этого программа выводит результат и останавливается.

```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);
    }
```

```c
#ifndef CALCULATE_H_
#define CALCULATE_H_
float Calculate(float Numeral,char Operation[4]);
#endif
```

```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;
}
```

3. Выполнил компиляцию программы посредством gcc:

```
nalukashov@dk5n55 ~/work/os/lab_prog $ gcc -c calculate.c
nalukashov@dk5n55 ~/work/os/lab_prog $ gcc -c main.c
main.c: В функции «main»:
main.c:11:11: предупреждение: формат «%s» ожидает аргумент типа «char *», но аргумент 2 имеет
rmat=]
   11 |   scanf("%s",&Operation);
      |          ~^   ~~~~~~~~~~
      |           |   |
      |           |   char (*)[4]
      |           char *
nalukashov@dk5n55 ~/work/os/lab_prog $ gcc calculate.o main.o -o calcul -lm
```

4. Создал 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 *~
```

В содержании файла указаны флаги компиляции, тип компилятора и файлы, которые должен собрать сборщик.

5. С помощью gdb выполнил отладку программы calcul (перед использованием gdb исправила Makefile): – запустите отладчик GDB, загрузив в него программу для отладки: gdb ./calcul – для запуска программы внутри отладчика ввела команду run

```
nalukashov@dk5n55 ~/work/os/lab_prog $ gdb ./calcul
GNU gdb (Gentoo 10.1 vanilla) 10.1
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-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://bugs.gentoo.org/>.
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...
(No debugging symbols found in ./calcul)
(gdb) run
Starting program: /afs/.dk.sci.pfu.edu.ru/home/n/a/nalukashov/work/os/lab_prog/calcul
Число: 2
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): *
Множитель: 2
  4.00
```

6. С помощью утилиты splint попробуйте проанализировать коды файлов

```
nalukashov@dk5n55 ~/work/os/lab_prog $ splint calculate.c
Splint 3.1.2 --- 13 Jan 2021

calculate.h:3:36: 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:7:32: Function parameter Operation declared as manifest array (size
                    constant is meaningless)
calculate.c: (in function Calculate)
calculate.c:13:7: 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:19:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:25:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:31:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:32:10: 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:35:14: Return value type double does not match declared type float:
                     (HUGE_VAL)
  To allow all numeric types to match, use +relaxtypes.
calculate.c:42:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:43:13: Return value type double does not match declared type float:
                     (pow(Numeral, SecondNumeral))
calculate.c:46:11: Return value type double does not match declared type float:
                     (sqrt(Numeral))
calculate.c:48:11: Return value type double does not match declared type float:
                     (sin(Numeral))
calculate.c:50:11: Return value type double does not match declared type float:
                     (cos(Numeral))
calculate.c:52:11: Return value type double does not match declared type float:
                     (tan(Numeral))
calculate.c:55:13: Return value type double does not match declared type float:
                     (HUGE_VAL)
```

calculate.c и main.c.

```
nalukashov@dk5n55 ~/work/os/lab_prog $ splint main.c
Splint 3.1.2 --- 13 Jan 2021

calculate.h:3:36: 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:9:3: 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:11:14: 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:11:11: Corresponding format code
main.c:11:3: Return value (type int) ignored: scanf("%s", &Ope...
```

#Контрольные вопросы

7. Информацию об этих программах можно получить с помощью функций

info и man.

8. 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. 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 заключается в компиляции всей программы в целом и получении исполняемого модуля.

5. 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. 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. Пример можно найти в задании 5.

7. 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. 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 выполнения вызываемых 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

9. 1) Выполнила компиляцию программы 2)Увидела ошибки в программе 3) Открыла редактор и исправила программу 4) Загрузила программу в отладчик gdb 5) run — отладчик выполнил программу, ввела требуемые значения. 6) Использовала другие команды отладчика и проверила работу программы

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

11. 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.

12. 1. Проверка корректности задания аргументов всех использованных в программе функций, а также типов возвращаемых ими значений;

13. Поиск фрагментов исходного текста, корректных с точки зрения синтаксиса языка

Си, но малоэффективных с точки зрения их реализации или содержащих в себе семантические ошибки;

3. Общая оценка мобильности пользовательской программы.

3 Вывод

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.