Front matter

lang: ru-RU title: "Отчет по лабораторной работе №14" subtitle: "по дисциплине: Операционные системы" author: "Трефилова Мария Андреевна"

Formatting

toc-title: "Содержание" toc: true # Table of contents toc_depth: 2 lof: false # List of figures lot: false # List of tables fontsize: 12pt linestretch: 1.5 papersize: a4paper documentclass: scrreprt polyglossia-lang: russian polyglossia-otherlangs: english mainfont: PT Serif romanfont: PT Serif sansfont: PT Sans monofont: PT Mono mainfontoptions: Ligatures=TeX romanfontoptions: Ligatures=TeX sansfontoptions: Ligatures=TeX,Scale=MatchLowercase monofontoptions: Scale=MatchLowercase indent: true pdf-engine: lualatex header-includes: - \linepenalty=10 # the penalty added to the badness of each line within a paragraph (no associated penalty node) Increasing the value makes tex try to have fewer lines in the paragraph. - \interlinepenalty=0 # value of the penalty (node) added after each line of a paragraph. - \hyphenpenalty=50 # the penalty for line breaking at an automatically inserted hyphen - \exhyphenpenalty=50 # the penalty for line breaking at an explicit hyphen - \binoppenalty=700 # the penalty for breaking a line at a binary operator - \relpenalty=500 # the penalty for breaking a line at a relation - \clubpenalty=150 # extra penalty for breaking after first line of a paragraph - \widowpenalty=150 # extra penalty for breaking before last line of a paragraph - \displaywidowpenalty=50 # extra penalty for breaking before last line before a display math - \brokenpenalty=100 # extra penalty for page breaking after a hyphenated line - \predisplaypenalty=10000 # penalty for breaking before a display - \postdisplaypenalty=0 # penalty for breaking after a display - \floatingpenalty = 20000 # penalty for splitting an insertion (can only be split footnote in standard LaTeX) - \raggedbottom # or \flushbottom - \usepackage{float} # keep figures where there are in the text

- \floatplacement{figure}{H} # keep figures where there are in the text

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.

Выполнение лабораторной работы

Начальный этап

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

```
[matrefilova@matrefilova os]$ cd ~/work/os/lab_prog
```

2. Создала 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.

```
bash: mcedit: команда не найдена...
[matrefilova@matrefilova lab_prog]$ touch calculate.h
[matrefilova@matrefilova lab_prog]$ cat calculate.h
[matrefilova@matrefilova lab_prog]$ cat calculate.h
[matrefilova@matrefilova lab_prog]$ cd ~/work/os
[matrefilova@matrefilova os]$ cd ~/work/os/lab_prog
[matrefilova@matrefilova lab_prog]$ mcedit calculate.h
bash: mcedit: команда не найдена...
[matrefilova@matrefilova lab_prog]$ vi calculate.h
[matrefilova@matrefilova lab_prog]$ vi calculate.h
[matrefilova@matrefilova lab_prog]$ touch main.c
[matrefilova@matrefilova lab_prog]$ vi main.c
[matrefilova@matrefilova lab_prog]$ vi calculate.h
[matrefilova@matrefilova lab_prog]$ touch calculate.c
[matrefilova@matrefilova lab_prog]$ vi calculate.c
[matrefilova@matrefilova lab_prog]$ vi calculate.c
[matrefilova@matrefilova lab_prog]$
```

Реализация функций калькулятора в файле calculate.c:

matrefilova@matrefilova:~/work/os/lab_prog

Файл Правка Вид Поиск Терминал Справка

```
#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("Second term: ");
        scanf("%f", &SecondNumeral);
        return(Numeral + SecondNumeral);
        }
else if(strncmp(Operation, "-", 1) == 0)
        {
        printf("Subtrahend: ");
        scanf("%f", &SecondNumeral);
        return(Numeral - SecondNumeral);
        }
else if(strncmp(Operation, "*", 1) == 0)
        {
        printf("Factor: ");
        scanf("%f", &SecondNumeral);
        return(Numeral * SecondNumeral);
        }
else if(strncmp(Operation, "/", 1) == 0)
-- ВСТАВКА --                                                    1,2       Наверху
```

matrefilova@matrefilova:~/work/os/lab_prog

Файл Правка Вид Поиск Терминал Справка

```
        scanf("%f", &SecondNumeral);
        if(SecondNumeral == 0)
        {
        printf("error");
        return(HUGE_VAL);
        }
        else
        return(Numeral / SecondNumeral);
        }
else if(strncmp(Operation, "pow", 3) ==0)
        {
        printf("Degree: ");
        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("Incorrectly");
        return(HIGE_VAL);
```

Интерфейсный файл calculate.h, описывающий формат вызова функции калькулятора:

matrefilova@matrefilova:~/work/os/lab_prog

Файл Правка Вид Поиск Терминал Справка

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

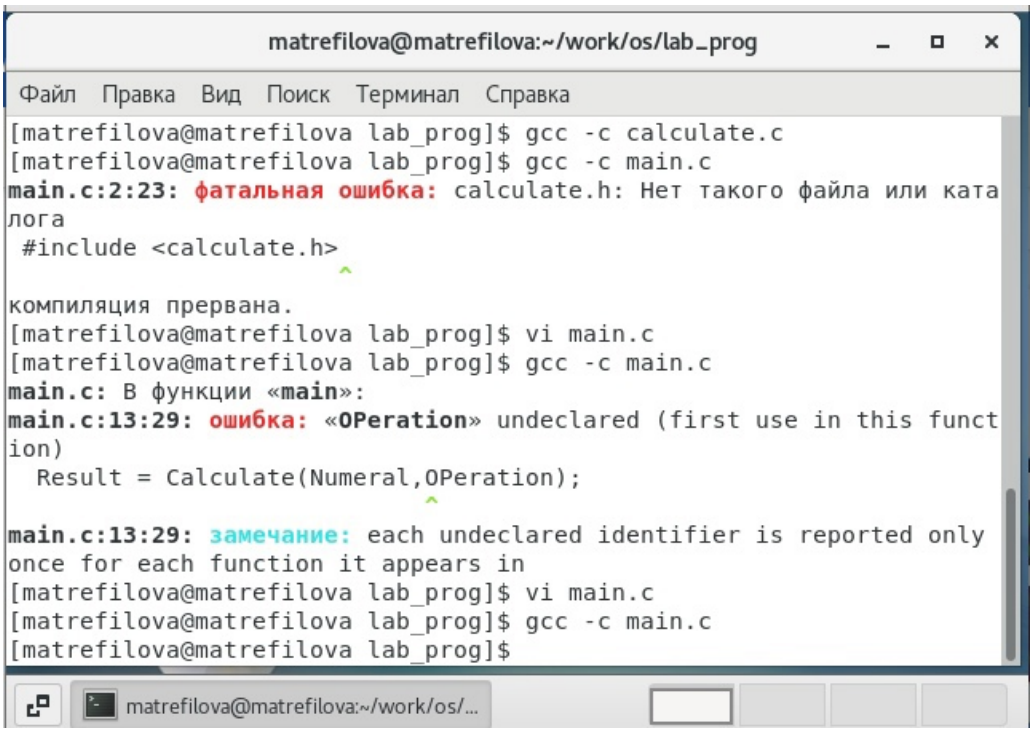
Основной файл main.c, реализующий интерфейс пользователя к калькулятору:

matrefilova@matrefilova:~/work/os/lab_prog

Файл Правка Вид Поиск Терминал Справка

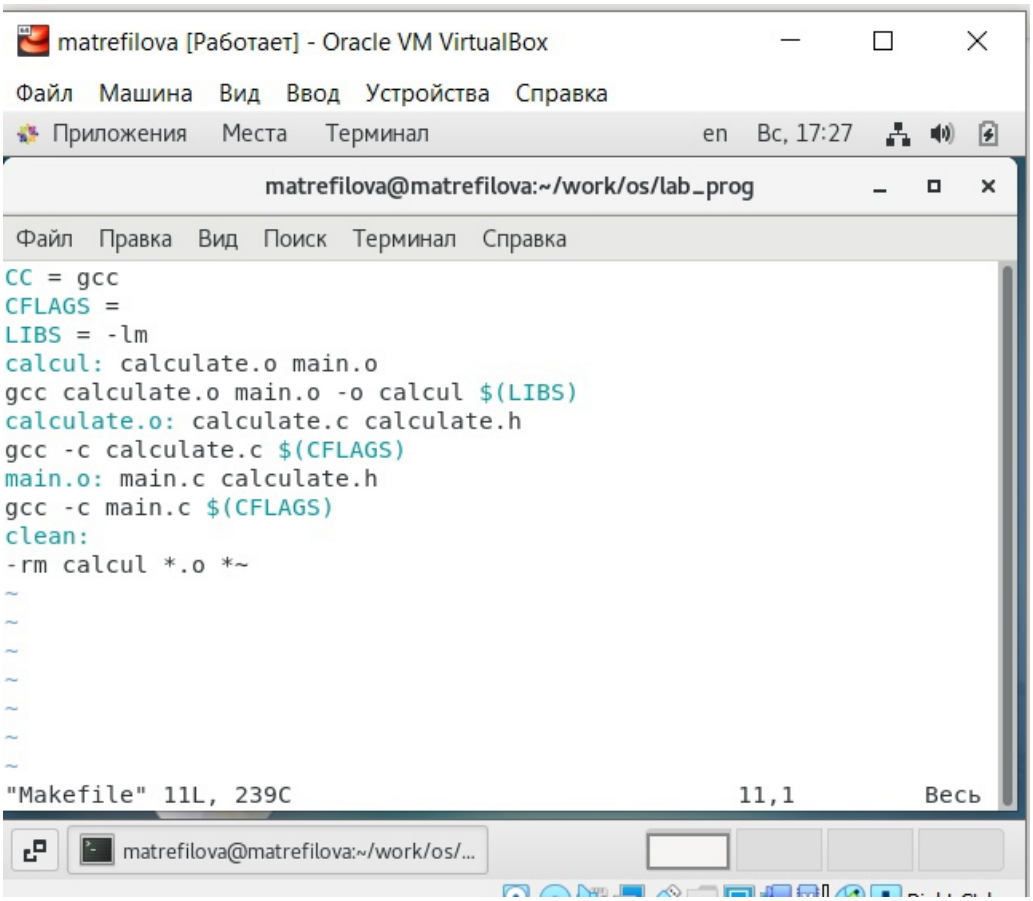
```
#include <stdio.h>
#include <calculate.h>
int
main (void)
{
        float Numeral;
        char Operation[4];
        float Result;
        printf("Numeral: ");
        scanf("%f", &Numeral);
        printf("Operation (+,-,*,/,pow,sqrt,sin,cos,tan): ");
        scanf("%s", &Operation);
        Result = Calculate(Numeral,OPeration);
        printf("%6.2f\n", Result);
        return 0;
}
:wq
```

3.Выполнила компиляцию программы посредством gcc:




```
[matrefilova@matrefilova lab_prog]$ gcc -c main.c
[matrefilova@matrefilova lab_prog]$ gcc calculate.o main.o -o calcul
-lm
[matrefilova@matrefilova lab_prog]$ make Makefile
```

4. Исправила синтаксические ошибки.
5. Создала Makefile.



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

6. С помощью gdb выполнила 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

```
(gdb) run
Starting program:
No executable file specified.
Use the "file" or "exec-file" command.
(gdb)
```

– для постраничного (по 9 строк) просмотра исходного код использовала команду list

– для просмотра строк с 12 по 15 основного файла использовала list с параметрами:

list 12,15

```
(gdb) list
No symbol table is loaded.  Use the "file" command.
(gdb) list calculate.c:20,29
No symbol table is loaded.  Use the "file" command.
(gdb)
```

– запустила программу внутри отладчика и убедилась, что программа остановится в момент прохождения точки останова

– отладчик выдал следующую информацию, а команда backtrace показала весь стек вызываемых функций от начала программы до текущего места:

– посмотрела, чему равно на этом этапе значение переменной Numeral, введя:

print Numeral

– сравнила с результатом вывода на экран после использования команды:

display Numeral

– убрала точки останова - С помощью утилиты splint попробуйте проанализировать коды файлов

calculate.c и main.c.

Выводы

Таким образом, 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.

Контрольные вопросы

1. Информацию об этих программах можно получить с помощью функций info и man.
2. 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;

    2) 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.
```