РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ

Факультет физико-математических и естественных наук

Кафедра прикладной информатики и теории вероятностей

ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ № 14

дисциплина: Операционные системы

Студент: Понкратова Христина Анатольевна

Группа: НПМбд-02-20

МОСКВА

2021 г.

Цель работы:

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

Ход работы:

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

```
khristina@kaponkratova:~$ mkdir work
khristina@kaponkratova:~$ cd work
khristina@kaponkratova:~/work$ mkdir os
khristina@kaponkratova:~/work$ cd os
khristina@kaponkratova:~/work/os$ mkdir lab_prog
khristina@kaponkratova:~/work/os$ cd lab_prog
```

2. Создали в нём файлы: calculate.h, calculate.c, main.c.

```
khristina@kaponkratova:~/work/os/lab_prog$ touch calculate.h calculate.c main.c
khristina@kaponkratova:~/work/os/lab_prog$ ls
calculate.c  calculate.h  main.c
khristina@kaponkratova:~/work/os/lab_prog$
```

Реализация функций калькулятора в файле calculate.c:

```
1 ////////////////////////////////////
2 // calculate.c
3 #include <stdio.h>
4 #include <math.h>
5 #include <string.h>
6 #include "calculate.h"
7 float
8 Calculate(float Numeral, char Operation[4])
9 {
10 float SecondNumeral;
11 if(strncmp(Operation, "+", 1) == 0)
12 {
13 printf("Второе слагаемое: ");
14 scanf("%f",&SecondNumeral);
15 return(Numeral + SecondNumeral);
16 }
17 else if(strncmp(Operation, "-", 1) == 0)
18 {
19 printf("Вычитаемое: ");
20 scanf("%f",&SecondNumeral);
   meral - SecondNumeral);
   trncmp(Operation, "*", 1) == 0)
   ножитель: ");
   ",&SecondNumeral);
   meral * SecondNumeral);
29 else if(strncmp(Operation, "/", 1) == 0)
30 {
31 printf("Делитель: ");
32 scanf("%f",&SecondNumeral);
33 if(SecondNumeral == 0)
34 {
35 printf("Ошибка: деление на ноль! ");
36 return(HUGE_VAL);
37 }
```

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

Открыть | *calculate.c ~/work/os/lab_prog | Сохранить

```
1 ////////////////////////////////////
2 // calculate.h
3 #ifndef CALCULATE_H_
4 #define CALCULATE_H_
5 float Calculate(float Numeral, char Operation[4]);
6 #endif /*CALCULATE_H_*/
```

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

Открыть *main.c ~/work/os/lab_prog Сохр

```
////////////////////////////////////////
// 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;
}
```

3. Выполнили компиляцию программы посредством gcc:

```
khristina@kaponkratova:~/work/os/lab_prog$ gedit calculate.h
khristina@kaponkratova:~/work/os/lab_prog$ gedit main.c
khristina@kaponkratova:~/work/os/lab_prog$ gcc -c calculate.c
khristina@kaponkratova:~/work/os/lab_prog$
```

4. В файле main.c допущена ошибка в строке «scanf("%s",&Operation);», не нужен &.
5. Создали Makefile со следующим содержанием:

Открыть *makefile ~/work/os/lab_prog

```
#
# 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 содерании файла указаны флаги компиляции, тип компилятора и файлы, которые должен собрать сборщик.

6. Выполнили отладку программы calcul. Для отладки используем gdb.

7. С помощью утилиты splint проанализировали коды файлов calculate.c и main.c.

Вывод:

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.

Ответы на контрольные вопросы:

1. Дополнительную информацию о 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. 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. 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 – 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;
9. 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. 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. 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.