РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ

Факультет Физико-Математических Наук

Кафедра прикладной информатики и теории вероятностей

ОТЧЁТ

ПО ЛАБОРАТОРНОЙ РАБОТЕ № 14

дисциплина: Операционные системы

Студент: Манаева Варвара Евгеньевна

Группа: НФИбд-01-20

Преподаватель: Кулябов Дмитрий Сергеевич

МОСКВА

2021 г.

Техническое оснащение:

- Персональный компьютер с операционной системой Windows 7;
- Планшет для записи видеосопровождения и голосовых комментариев;
- Виртуальная коробка VirtualBox, виртуальная машина с установленной на ней операционной системой CentOS;
- Microsoft Teams, использующийся для записи скринкаста лабораторной работы;
- Приложение MarkPad 2 для редактирования файлов формата *md*;
- *pandoc* для конвертации файлов отчётов и презентаций.

Объект и предмет исследования: средства, применяемые при разработке программного обеспечения в ОС типа UNIX/Linux.

Цель [1]: 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.

Задачи:

1) Научиться разрабатывать приложения в ОС типа Linux/Unix;
2) Приобрести навыки анализа, тестирования и отладки приложений в ОС типа Linux/Unix;
3) Создать калькулятор с простейшими функциями на языке программирования С.

Теоретические вводные данные [1]:

1) Этапы разработки приложений

Процесс разработки программного обеспечения обычно разделяется на следующие этапы:

- планирование, включающее сбор и анализ требований к функционалу и другим характеристикам разрабатываемого приложения;
- проектирование, включающее в себя разработку базовых алгоритмов и спецификаций, определение языка программирования;
- непосредственная разработка приложения:
- кодирование — по сути создание исходного текста программы (возможно в нескольких вариантах);
- анализ разработанного кода;
- сборка, компиляция и разработка исполняемого модуля;
- 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.

2) Компиляция исходного текста и построение исполняемого файла

Стандартным средством для компиляции программ в ОС типа UNIX является GCC (GNU Compiler Collection). Это набор компиляторов для разного рода языков программирования (C, C++, Java, Фортран и др.). Работа с GCC производится при помощи одноимённой управляющей программы gcc, которая интерпретирует аргументы командной строки, определяет и осуществляет запуск нужного компилятора для входного файла.
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.
Для компиляции файла main.c, содержащего написанную на языке C простейшую программу:

```
/*
* main.c
*/
#include <stdio.h>
int main()
{
printf("Hello World!\n");
return 0;
}
```

достаточно в командной строке ввести:

```
gcc -c main.c
```

Таким образом, 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
```

Описание некоторых опций gcc приведено в табл. 11.1.

Таблица 11.1

Некоторые опции компиляции в gcc

Опция	Описание
`-c`	компиляция без компоновки — создаются объектные файлы `file.o`
`-o file-name`	задать имя `file-name` создаваемому файлу
`-g`	поместить в файл (объектный или исполняемый) отладочную информацию для отладчика `gdb`
`-MM`	вывести зависимости от заголовочных файлов C и/или C++ программ в формате, подходящем для утилиты `make`; при этом объектные или исполняемые файлы не будет созданы
`-Wall`	вывод на экран сообщений об ошибках, возникших во время компиляции

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. 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:

```
<цель_1> <цель_2> ... : <зависимость_1> <зависимость_2> ...
    <команда 1>
    ...
    <команда n>
```

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.

Рассмотрим пример Makefile для написанной выше простейшей программы, выводящей на экран приветствие ’Hello World!’:

```
hello: main.c
gcc -o hello main.c
```

Здесь в первой строке hello — цель, main.c — название файла, который мы хотим скомпилировать; во второй строке, начиная с табуляции, задана команда компиляции gcc с опциями.

Для запуска программы необходимо в командной строке набрать команду make:

```
make
```

Общий синтаксис Makefile имеет вид:

```
target1 [target2...]:[:] [dependment1...]
        [(tab)commands] [#commentary]
        [(tab)commands] [#commentary]
```

Здесь знак # определяет начало комментария (содержимое от знака # и до конца строки не будет обрабатываться. Одинарное

двоеточие указывает на то, что последовательность команд должна содержаться в одной 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.

3) Тестирование и отладка

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
```

Затем можно использовать по мере необходимости различные команды gdb. Наиболее часто используемые команды gdb приведены в табл. 11.2.

Таблица 11.2

Некоторые команды gdb

Команда	Описание действия
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

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.

4) Анализ исходного текста программы

Ещё одним средством проверки исходных кодов программ, написанных на языке C, является утилита splint. Эта утилита 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 работе программы, переменные с некорректно заданными значениями и типами и многое другое.

Рассмотрим следующий небольшой пример:

```
float sum(float x, float y){
return x + y;
}
int main(){
int x, y;
float z;
x = 10;
y = 12;
z = x + y + sum(x, y);
return z;
}
```

Сохраните код данного примера в файл example.c. Для анализа кода программы следует выполнить следующую команду:

```
splint example.c
```

В результате на экран будут выведены следующие пять предупреждений

```
Splint 3.1.2 --- 03 May 2009
example.c: (in function main)
example.c:10:18: Function sum expects arg 1 to be float gets int: x
To allow all numeric types to match, use +relaxtypes.
example.c:10:21: Function sum expects arg 2 to be float gets int: y
example.c:10:2: Assignment of int to float: z = x + y + sum(x, y)
example.c:11:9: Return value type float does not match declared
type int: z
Finished checking --- 4 code warnings
```

Первые два предупреждения относятся к функции sum, которая определена для двух аргументов вещественного типа, но при вызове ей передаются два аргумента x и y целого типа. Третье предупреждение относится к присвоению вещественной переменной z значения целого типа, которое получается в результате суммирования x + y + sum(x, y). Далее вещественное число z возвращается в качестве результата работы функции main, хотя данная функция объявлена как функция, возвращающая целое значение.

В качестве упражнения попробуйте скомпилировать программу компилятором gcc и сравнить выдаваемые им предупреждения с приведёнными выше.

Этапы работы:

1) В домашнем каталоге создайте подкаталог ~/work/os/lab_prog.

!

2) Создайте в нём файлы: 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.

Реализация функций калькулятора в файле calculate.h:

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

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

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

```
#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:

```
//////////////////////////////////////
// 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) Выполните компиляцию программы посредством gcc: gcc -c calculate.c gcc -c main.c gcc calculate.o main.o -o calcul -lm

!

4) При необходимости исправьте синтаксические ошибки.

!

5) Создайте Makefile со следующим содержанием:

```
#
# 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 нём указаны :

- CC - тип компилятора.
- CFLAGS - опции, с которыми мы компилируем основной файл.
- CompFlags - опции, с которыми мы компилируем остальные файлы.
- Компиляция каждого зависящего файла .o
- Сборщик всей программы "calculate"
- Команда clean, для быстрого удаления всех файлов.

6) С помощью gdb выполните отладку программы calcul (перед использованием gdb исправьте Makefile):

Запустите отладчик GDB, загрузив в него программу для отладки:

```
gdb ./calcul
```

!

Для запуска программы внутри отладчика введите команду run:

```
run
```

!

Для постраничного (по 9 строк) просмотра исходного код используйте команду list:

```
list
```

!

Для просмотра строк с 12 по 15 основного файла используйте list с параметрами:

```
list 12,15
```

!

Для просмотра определённых строк не основного файла используйте list с параметрами:

```
list calculate.c:20,29
```

!

Установите точку останова в файле calculate.c на строке номер 21:

```
list calculate.c:20,27
break 21
```

!

Выведите информацию об имеющихся в проекте точка останова:

```
info breakpoints
```

!

Запустите программу внутри отладчика и убедитесь, что программа остановится в момент прохождения точки останова:

```
run
5-
backtrace
```

!

Отладчик выдаст следующую информацию:

```
#0 Calculate (Numeral=5, Operation=0x7fffffffd280 "-")
at calculate.c:21
#1 0x0000000000400b2b in main () at main.c:17
```

а команда backtrace покажет весь стек вызываемых функций от начала программы до текущего места.

!

Посмотрите, чему равно на этом этапе значение переменной Numeral, введя:

```
print Numeral
```

На экран должно быть выведено число 5.

!

Сравните с результатом вывода на экран после использования команды:

```
display Numeral
```

!

Уберите точки останова:

```
info breakpoints
delete 1
```

!

7) С помощью утилиты splint попробуйте проанализировать коды файлов calculate.c и main.c.

!

Выводы: 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]:

1) Как получить информацию о возможностях программ gcc, make, gdb и др.?

Ответ: Информацию об этих программах можно получить с помощью функций --help и man.

2) Назовите и дайте краткую характеристику основным этапам разработки приложений в UNIX.

Ответ: Разработка приложений в UNIX:

1. создание исходного кода программы;(файл с необходимым расширением(и кодом)).
2. сохранение различных вариантов исходников;
3. анализ исходников; необходимо отслеживать изменения исходного кода.
4. компиляция исходников и построение исполняемого модуля;
5. тестирование и отладка; (проверка кода на наличие ошибок)
6. сохранение всех изменений, выполняемых при тестировании и отладке.
7. Загрузка версии исходников в систему контроля версий GIT.

3) Что такое суффикс в контексте языка программирования? Приведите примеры использования.

Ответ: Суффикс определяет какая компиляция требуется. Суффиксы и префиксы указывают тип объекта.(пример main.c для компиляции на языке С)

4) Каково основное назначение компилятора языка С в UNIX?

Ответ: Основное назначение этого компилятора заключается в компиляции всей программы и получении исполняемого модуля.

5) Для чего предназначена утилита make?

Ответ: Утилита make нужна для исполнения команд из Makefile(-ов). (компиляции, очистки и тп).

6) Приведите пример структуры Makefile. Дайте характеристику основным элементам этого файла.

Ответ: Пример:

```
CC = compiler
CFLAGS = compiler flags( like -c or -g)
.....( and so on)
```

```
target1: dependencies ( for example: report: dependentFile.o ... dependentFile.o)
target2: dependencies
...
targetn: dependencies
<tab>(necessary) $(CC) dependentFile.o $(CFLAGS) report.
```

пример из лабараторной работы:

```
CC = g++
CFLAGS = -g -c
CompFlags= -g -o

calculate: Calculator.o main.o operation.o
    $(CC) Calculator.o main.o operation.o $(CompFlags) calculate

Calculator.o: Calculator.cpp CalcHeader.h
    $(CC) $(CFLAGS) Calculator.cpp

main.o: main.cpp CalcHeader.h
    $(CC) $(CFLAGS) main.cpp

operation.o: operation.cpp CalcHeader.h
    $(CC) $(CFLAGS) operation.cpp

clean:
    rm -rf *.o calculate
```

7) Назовите основное свойство, присущее всем программам отладки. Что необходимо сделать, чтобы его можно было использовать?

Ответ: 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.

Ответ: Смотри таблицу 11.2 [1].

Таблица 11.2

Некоторые команды gdb

Команда	Описание действия
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) Опишите по шагам схему отладки программы, которую Вы использовали при выполнении лабораторной работы.

Ответ:

```
1) Запустила Makefile (компиляция и сборка);
2) Начала отладку (run);
3) Вывела содержимое основного файла и Calculator.cpp;
4) Установила точку останова в Calculator.cpp.
5) Продолжила выполнение (run);
6) Посмотрела используемые функции (на данный момент времени  backtrace);
7) Использовала команды print & display;
8) Удалила точку останова;
9) Закончила отладку (ошибок не было найдено).
```

10) Прокомментируйте реакцию компилятора на синтаксические ошибки в программе при его первом запуске.

Ответ: Реакция была нормальной (ошибок не было)

11) Назовите основные средства, повышающие понимание исходного кода программы.

Ответ: cppcheck; splint; cscope;

12) Каковы основные задачи, решаемые программой splint?

Ответ: 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.

Библиография:

1. Текст лабораторной работы № 14.