

Отчёт по лабораторной работе №13

Дисциплина: Операционные системы

Елизавета Андреевна Алмазова

Содержание

| | |
|---|---|
| Цель работы | 1 |
| Задание | 1 |
| Теоретическое введение | 2 |
| Этапы разработки приложений | 2 |
| Компиляция исходного текста и построение исполняемого файла | 2 |
| Тестирование и отладка | 4 |
| Анализ исходного текста программы | 4 |
| Выполнение лабораторной работы | 5 |
| Выводы | 8 |
| Ответы на контрольные вопросы | 8 |

Цель работы

Цель данной лабораторной работы - приобрести простейшие навыки разработки, анализа, тестирования и отладки приложений в ОС типа UNIX/Linux на примере создания на языке программирования C калькулятора с простейшими функциями.

Задание

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.
3. Выполните компиляцию программы посредством `gcc`.
4. При необходимости исправьте синтаксические ошибки.
5. Создайте `Makefile`. Поясните в отчете его содержание.
6. С помощью `gdb` выполните отладку программы `calcul` (перед использованием `gdb` исправьте `Makefile`):
 - Запустите отладчик GDB, загрузив в него программу для отладки
 - Для запуска программы внутри отладчика введите команду `run`

- Для постраничного (по 9 строк) просмотра исходного код используйте команду `list`
 - Для просмотра строк с 12 по 15 основного файла используйте `list` с параметрами
 - Для просмотра определённых строк не основного файла используйте `list` с параметрами
 - Установите точку останова в файле `calculate.c` на строке номер 21
 - Выведите информацию об имеющихся в проекте точка останова:
 - Запустите программу внутри отладчика и убедитесь, что программа остановится в момент прохождения точки останова
 - Посмотрите, чему равно на этом этапе значение переменной `Numeral`
 - Сравните с результатом вывода на экран после использования команды: `display Numeral`
 - Уберите точки останова
7. С помощью утилиты `splint` попробуйте проанализировать коды файлов `calculate.c` и `main.c`.

Теоретическое введение

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.

Компиляция исходного текста и построение исполняемого файла

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 `gsc`, 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. 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.

Для сборки разрабатываемого приложения и собственно компиляции полезно воспользоваться утилитой make. В самом простом случае Makefile имеет следующий синтаксис:

```
...: ...  
<команда 1>  
...
```

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.

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 normally  
  
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.

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

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.

Выполнение лабораторной работы

1. В домашнем каталоге создала подкаталог `~/work/os/lab_prog` с помощью команд `cd work` (переход в существующую папку), `mkdir os`, `mkdir os/lab_prog` (создание подкаталога).
2. Создала в нём файлы: `calculate.c` (рис.1), `calculate.h` (рис.2), `main.c` (рис.3). 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.

```
1 //////////////////////////////////////////////////
2 // calculate.c
3
4
5 #include <stdio.h>
6 #include <math.h>
7 #include <string.h>
8 #include "calculate.h"
9
10 float Calculate(float Numeral, char Operation[4])
11 {
12     float SecondNumeral;
13     if(strncmp(Operation, "+", 1) == 0) {
14         printf("Второе слагаемое: ");
15         scanf("%f", &SecondNumeral);
16         return(Numeral + SecondNumeral);
17     }
18     else if(strncmp(Operation, "-", 1) == 0) {
19         printf("Вычитаемое: ");
20         scanf("%f", &SecondNumeral);
21         return(Numeral - SecondNumeral);
22     }
23     else if(strncmp(Operation, "*", 1) == 0) {
24         printf("Множитель: ");
25         scanf("%f", &SecondNumeral);
26         return(Numeral * SecondNumeral);
27     }
28     else if(strncmp(Operation, "/", 1) == 0) {
29         printf("Делитель: ");
30         scanf("%f", &SecondNumeral);
31         if(SecondNumeral == 0) {
32             printf("Ошибка: деление на ноль! ");
33             return(HUGE_VAL);
34         }
35         else return(Numeral / SecondNumeral);
36     }
37     else if(strncmp(Operation, "pow", 3) == 0) {
38         printf("Степень: ");
39         scanf("%f", &SecondNumeral);
40         return(pow(Numeral, SecondNumeral));
41     }
42     else if(strncmp(Operation, "sqrt", 4) == 0) return(sqrt(Numeral));
43     else if(strncmp(Operation, "sin", 3) == 0) return(sin(Numeral));
44     else if(strncmp(Operation, "cos", 3) == 0) return(cos(Numeral));
45     else if(strncmp(Operation, "tan", 3) == 0) return(tan(Numeral));
46     else {
47         printf("Неправильно введено действие. ");
48         return(HUGE_VAL);
49     }
```

Рисунок 1 - `calculate.c`.

```

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

#ifndef CALCULATE_H_
#define CALCULATE_H_

float Calculate(float Numeral, char Operation[4]);

#endif /*CALCULATE_H_*/

```

Рисунок 2 - calculate.h.

```

1 ///////////////////////////////////////////////////
2 // main.c
3
4 #include <stdio.h>
5 #include "calculate.h"
6
7 int main(){
8     float Numeral;
9     char Operation[4];
10    float Result;
11    printf("Число: ");
12    scanf("%f",&Numeral);
13    printf("Операция (+,-,*,/,pow,sqrt,sin,cos,tan): ");
14    scanf("%s",&Operation);
15    Result = Calculate(Numeral, Operation);
16    printf("%.2f\n",Result);
17    return 0;
18 }

```

Рисунок 3 - main.c.

4. Выполнила компиляцию программы посредством gcc (gcc -c calculate.c, gcc -c main.c, gcc calculate.o main.o -o calcul -lm). Добавила пропущенные точки с запятой после предупреждения компилятора.
5. Создала Makefile. Переменная CC содержит название утилиты gcc, CFLAGS содержит опции утилиты, LIBS - опции для объединения объектных файлов в исполняемый. Имеются 4 цели: clean для удаления созданных исполняемых и объектных файлов, calcul для компиляции исполняемого файла, calculate.o и main.o для создания соответствующих объектных файлов.
6. С помощью gdb выполнила отладку программы calcul (перед использованием gdb исправила Makefile, добавив опцию -g и заменив gcc на переменную, рис.4, рис.5, рис.6):
 - Запустила отладчик GDB, загрузив в него программу для отладки: gdb ./calcul
 - Для запуска программы внутри отладчика ввела команду run
 - Для постраничного (по 9 строк) просмотра исходного кода использовала команду list
 - Для просмотра строк с 12 по 15 основного файла использовала list с параметрами: list 12,15
 - Для просмотра определённых строк не основного файла использовала list с параметрами: list calculate.c:20,29 (рис.7)
 - Установила точку останова в файле calculate.c на строке номер 21: list calculate.c:20,27, break 21
 - Вывела информацию об имеющихся в проекте точках останова: info breakpoints

- Запустила программу внутри отладчика (run, 5, -, backtrace) и убедилась, что программа остановится в момент прохождения точки останова.
- Отладчик выдал следующую информацию: #0 Calculate (Numeral=5, Operation=0x7fffffff280 "-") at calculate.c:21, #1 0x0000000000400b2b in main () at main.c:17, а команда backtrace показала весь стек вызываемых функций от начала программы до текущего места. Numeral на этом этапе равно 5: print Numeral и display Numeral показывают одинаковые значения. Затем убрала точки останова с помощью команды delete 1. (рис.8)

```
#
# Makefile
#

CC = gcc
CFLAGS = -g
LIBS = -lm

calcul: calculate.o main.o
$(CC) calculate.o main.o -o calcul $(LIBS)

calculate.o: calculate.c calculate.h
$(CC) -c calculate.c $(CFLAGS)

main.o: main.c calculate.h
$(CC) -c main.c $(CFLAGS)

clean:
-rm calcul *.o *~

# End Makefile
```

Рисунок 4 - Отредактированный Makefile.








| | |
|---|-----------|
|  calcul | 27.9 kB |
|  calculate.c | 1.5 kB |
|  calculate.h | 162 bytes |
|  calculate.o | 7.1 kB |
|  main.c | 352 bytes |
|  main.o | 4.6 kB |
|  Makefile | 290 bytes |

Рисунок 5 - Результат работы Makefile по компилированию программы.

```
[eaalmazova@fedora lab_prog]$ ./calcul
Число: 20
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): -
Вычитаемое: 15
5.00
[eaalmazova@fedora lab_prog]$
```

Рисунок 6 - Работа скомпилированной программы.

```
(gdb) list calculate.c:20,29
20         scanf("%f",&SecondNumeral);
21         return Numeral - SecondNumeral;
22     }
23     else if(strncmp(Operation,"+",1)==0) {
24         printf("Делитель: ");
25         scanf("%f",&SecondNumeral);
26         return Numeral - SecondNumeral;
27     }
28     else if(strncmp(Operation,"/",1)==0) {
29         printf("");
```

Рисунок 7 - list calculate.c:20,29.

```
Breakpoint 1, Calculate (Numeral=5, Operation=0x7fffffffdf44 "-") at calculate.c:21
:21
21         return Numeral - SecondNumeral;
(gdb) print Numeral
$1 = 5
(gdb) display Numeral
1: Numeral = 5
(gdb) info breakpoints
Num  Type      Disp Enb Address            What
1     breakpoint keep y  0x0000000000401282  in Calculate
                                     at calculate.c:21
      breakpoint already hit 1 time
(gdb) delete 1
```

Рисунок 8 - Работа с точкой останова.

7. С помощью утилиты splint проанализировала коды файлов calculate.c и main.c. Выяснились например такие факты, что в перечисленных файлах есть функция scanf, которая возвращает целое значение, но эти числа нигде не используются и не сохраняются. Также есть потеря данных при записи в переменную типа float и сравнение вещественного числа с нулем.

Выводы

В ходе выполнения данной лабораторной работы я приобрела простейшие навыки разработки, анализа, тестирования и отладки приложений в ОС типа UNIX/Linux на примере создания на языке программирования C калькулятора с простейшими функциями.

Ответы на контрольные вопросы

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

С помощью команды man или опции -help для каждой команды.

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

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.

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

Суффикс - расширение. 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. 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.

4. Каково основное назначение компилятора языка C в UNIX?э

Компиляция программы и получение исполняемого файла/модуля.

5. Для чего предназначена утилита make?

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

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

В самом простом случае Makefile имеет следующий синтаксис:

```
...: ...
<команда 1>
...
```

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.

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.

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

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

8. Назовите и дайте основную характеристику основным командам отладчика gdb.
- 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.
- Запустите отладчик GDB, загрузив в него программу для отладки
 - Для запуска программы внутри отладчика введите команду run
 - Для постраничного (по 9 строк) просмотра исходного кода используйте команду list
 - Для просмотра строк с 12 по 15 основного файла используйте list с параметрами

- Для просмотра определённых строк не основного файла используйте list с параметрами
 - Установите точку останова в файле calculate.c на строке номер 21
 - Выведите информацию об имеющихся в проекте точка останова:
 - Запустите программу внутри отладчика и убедитесь, что программа остановится в момент прохождения точки останова
 - Посмотрите, чему равно на этом этапе значение переменной Numeral
 - Сравните с результатом вывода на экран после использования команды: display Numeral
 - Уберите точки останова
10. Прокомментируйте реакцию компилятора на синтаксические ошибки в программе при его первом запуске.

In function ... expected ... token - я пропустила точку с запятой, и компилятор мне об этом сообщил.

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

escore для исследования функций в программе и lint для критической проверки программ, написанных на языке Си.

12. Каковы основные задачи, решаемые программой 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.