

Лабораторная работа №14

Дисциплина: Операционные системы

Галиев Казиз Жарылкасымович

Содержание

Цель работы	5
Выполнение лабораторной работы	6
Выводы	14
Контрольные вопросы	15
Список литературы	20

Список иллюстраций

0.1	В домашнем каталоге создаем подкаталог ~ work/os/lab_prog . . .	6
0.2	Создаем файл calculate.c	6
0.3	продолжение файла calculate.c	7
0.4	продолжение файла calculate.c	7
0.5	Создадим файл calculate.h	8
0.6	Создадим файл main.c	8
0.7	Выполним компиляцию программы посредством gcc	9
0.8	Создадим Makefile	9
0.9	Запустим отладчик и программу внутри отладчика	10
0.10	Выполним программу	10
0.11	Просмотрим исходную программу	11
0.12	Просмотрим определенные строки не основного файла, установим точку	11
0.13	Выведем информацию об имеющихся в проекте точках останова .	12
0.14	убедимся что программа остановилась в нужный момент	12
0.15	значение переменной Numeral, уберем точку останова	12
0.16	Проанализируем программный код с помощью утилиты splint . .	13

Список таблиц

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

1. В домашнем каталоге создаем подкаталог ~ work/os/lab_prog (рис. @fig:001)

```
[kzgaliev@fedora os]$ mkdir lab_prog
[kzgaliev@fedora os]$ ls
labos  lab_prog
[kzgaliev@fedora os]$
```

Рис. 0.1: В домашнем каталоге создаем подкаталог ~ work/os/lab_prog

2. Создаем файл calculate.c (рис. @fig:002) .

```
// 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("Умножаем: ");
        scanf("%f", &SecondNumeral);
        return(Numeral * SecondNumeral);
    }
    else if(strncmp(Operation, "/", 1) == 0)
    {
        printf("Делим: ");
        scanf("%f", &SecondNumeral);
        return(Numeral / SecondNumeral);
    }
    return(0);
}

-- INSERT --
```

Рис. 0.2: Создаем файл calculate.c

3. продолжение файла calculate.c (рис. @fig:003) .

```

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)
-- INSERT --

```

Рис. 0.3: продолжение файла calculate.c

4. продолжение файла calculate.c (рис. @fig:004) .

```

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);
}
}
}
-- INSERT --

```

Рис. 0.4: продолжение файла calculate.c

5. Создадим файл calculate.h (рис. @fig:005) .


```
[kzgaliev@fedora lab_prog]$ gcc -c calculate.c -g
[kzgaliev@fedora lab_prog]$ gcc -c main.c -g
main.c: В функции «main»:
main.c:17:1: ошибка: expected declaration or statement at end of input
17 | return 0;
    |
[kzgaliev@fedora lab_prog]$ vi main.c
[kzgaliev@fedora lab_prog]$ gcc -c main.c -g
[kzgaliev@fedora lab_prog]$ gcc calculate.o main.o -o calcul -lm
[kzgaliev@fedora lab_prog]$
```

Рис. 0.7: Выполним компиляцию программы посредством gcc

8. Создадим Makefile (рис. @fig:008) .

```
kzgaliev@fedora:~/work/os/lab_prog — /usr/libexe...
#
# 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
# End Makefile
```

Рис. 0.8: Создадим Makefile

9. Запустим отладчик и программу внутри отладчика (run) (рис. @fig:009) .

```
[kzgaliev@fedora lab_prog]$ gdb ./calcul
GNU gdb (GDB) Fedora Linux 13.1-3.fc37
Copyright (C) 2023 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-redhat-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
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...
(gdb) run
Starting program: /home/kzgaliev/work/os/lab_prog/calcul

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.fedoraproject.org/>
Enable debuginfod for this session? (y or [n])
```

Рис. 0.9: Запустим отладчик и программу внутри отладчика

10. Выполним программу (рис. @fig:010) .

```
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
Downloading separate debug info for system-supplied DSO at 0x7ffff7fc6080
Downloading separate debug info for /lib64/libm.so.6
Downloading separate debug info for /lib64/libc.so.6
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Число: 2
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): +
Второе слагаемое: 3
5.00
[Inferior 1 (process 3908) exited normally]
(gdb)
```

Рис. 0.10: Выполним программу

11. Просмотрим исходную программу (list), посмотрим строки с 12 по 15 (рис. @fig:011) .

```

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

```

Рис. 0.11: Просмотрим исходную программу

12. Просмотрим определенные строки не основного файла, установим точку останова 21 строке (рис. @fig:012) .

```

(gdb) list calculate.c:20,29
20 scanf("%f",&SecondNumeral);
21 return(Numeral - SecondNumeral);
22 }
23 else if(strncmp(Operation, "-", 1) == 0)
24 {
25 printf("Множитель: ");
26 scanf("%f",&SecondNumeral);
27 return(Numeral * SecondNumeral);
28 }
29 else if(strncmp(Operation, "/", 1) == 0)
(gdb) list calculate.c:20,27
20 scanf("%f",&SecondNumeral);
21 return(Numeral - SecondNumeral);
22 }
23 else if(strncmp(Operation, "-", 1) == 0)
24 {
25 printf("Множитель: ");
26 scanf("%f",&SecondNumeral);
27 return(Numeral * SecondNumeral);
(gdb) break 21
Breakpoint 1 at 0x401234: file calculate.c, line 21.
(gdb)

```

Рис. 0.12: Просмотрим определенные строки не основного файла, установим точку

13. Выведем информацию об имеющихся в проекте точках останова (рис. @fig:013) .

```
(gdb) info breakpoints
Num  Type      Disp Enb Address      What
1    breakpoint keep y  0x0000000000401234  in calculate
                                     at calculate.c:21
```

Рис. 0.13: Выведем информацию об имеющихся в проекте точках останова

14. Запустим программу внутри отладчика и убедимся, что программа остановилась в момент прохождения точки останова (рис. @fig:014).

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/kzgaliev/work/os/lab_prog/calcul
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Число: 5
Операция (+, -, *, /, pow, sqrt, sin, cos, tan): -
Вычитаемое: backtrace

Breakpoint 1, calculate (Numeral=5, Operation=0x7fffffffde84 "-")
at calculate.c:21
21  return(Numeral - SecondNumeral);
(gdb) print Numeral
```

Рис. 0.14: убедимся что программа остановилась в нужный момент

15. Посмотрим, чему равно на этом этапе значение переменной Numeral разными способами, уберем точку останова (рис. @fig:015).

```
(gdb) print Numeral
$1 = 5
(gdb) display Numeral
1: Numeral = 5
(gdb) info breakpoints
Num  Type      Disp Enb Address      What
1    breakpoint keep y  0x0000000000401234  in calculate
                                     at calculate.c:21

breakpoint already hit 1 time
(gdb) delete 1
(gdb)
```

Рис. 0.15: значение переменной Numeral, уберем точку останова

16. Проанализируем программный код с помощью утилиты splint (рис. @fig:016).

```
[kzgaliev@fedora lab_prog]$ splint calculate.c
Splint 3.1.2 --- 23 Jul 2022

calculate.h:5:37: 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:8:31: Function parameter Operation declared as manifest array (size
constant is meaningless)
calculate.c: (in function Calculate)
calculate.c:14:1: 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:20:1: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:26:1: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:32:1: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:33:4: 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
```

Рис. 0.16: Проанализируем программный код с помощью утилиты splint

Выводы

В результате лабораторной работы я 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.

Контрольные вопросы

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

Более подробную информацию по работе с gdb можно получить с помощью команд `gdb -h` и `man gdb`.

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

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.

4. Каково основное назначение компилятора языка C в UNIX? Компилятор — это программа, которая переводит текст, написанный на языке программирования, в машинные коды. С помощью компиляторов компьютеры могут понимать разные языки программирования, в том числе высокоуровневые, то есть близкие к человеку и далекие от «железа». Процесс работы компилятора с кодом называется компиляцией, или сборкой. По сути, компилятор — комплексный «переводчик», который собирает, или компилирует, программу в исполняемый файл. Исполняемый файл — это набор инструкций для компьютера, который тот понимает и может выполнить.
5. Для чего предназначена утилита 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.

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.

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` Затем можно использовать по мере необходимости различные команды `gdb`.

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: `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 останавливается в момент прохождения точки останова: `run 5`
- `backtrace` – Посмотрите, чему равно на этом этапе значение переменной `Numeral`, введя: `print Numeral display Numeral` – Уберите точки останова: `info breakpoints delete 1`
10. Назовите основные средства, повышающие понимание исходного кода программы. 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: – `cscore` - исследование функций, содержащихся в программе; – `lint` – критическая проверка программ, написанных на языке Си.
 11. 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 С 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 много другого.

Список литературы