

Отчёт по лабораторной работе №13

Средства, применяемые при разработке программного обеспечения в
ОС типа UNIX/Linux.

Самигуллин Эмиль Артурович

1 Цель 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.

2 Задание

1. Написать приложение, выполняющее функции калькулятора на языке C.

3 Теоретическое введение

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.

4 Выполнение лабораторной работы

1. Создал каталог `~/work/os/lab_prog` (рис. 1).

```
[edsmirnovmaljce@edsmirnovmaljce ~]$ mkdir work/os/lab_prog
```

рис. 1: создание каталога `lab_prog`.

2. Написал на С программы, выполняющие функции калькулятора (рис. 2).

```
[edsmirnovmaljce@edsmirnovmaljce ~]$ touch calculate.h calculate.c main.c
[edsmirnovmaljce@edsmirnovmaljce ~]$ cd work/os/lab_prog
[edsmirnovmaljce@edsmirnovmaljce lab_prog]$ mcedit calculate.c

[edsmirnovmaljce@edsmirnovmaljce lab_prog]$ mcedit calculate.h

[edsmirnovmaljce@edsmirnovmaljce lab_prog]$ mcedit main.c
```

рис. 2: создание файлов.

3. Написал Makefile, компилирующий программы из предыдущего пункта и запустил утилиту `make` (рис. 3).

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

рис. 3: выполнение утилиты `make`.

4. С помощью `gdb` выполнил отладку `calcul`:

- Запустил отладчик (рис. 4).

```
[edsmirnovmaljce@edsmirnovmaljce ~]$ gdb work/os/lab_prog/calcul
GNU gdb (GDB) Fedora 11.2-2.fc35
Copyright (C) 2022 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 work/os/lab_prog/calcul...
```

рис. 4: запуск gdb.

- Запустил программу в отладчике (рис. 4).

```
(gdb) run
Starting program: /home/edsmirnovmaljce/work/os/lab_prog/calcul
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Число: 3
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): +
Второе слагаемое: 2
    5.00
[Inferior 1 (process 3527) exited normally]
```

рис. 4: запуск программы.

- Просмотрел первые 9 строк исходного кода (рис. 5).

```
(gdb) list
1      //////////////////////////////////////////
2      // main.c
3
4      #include <stdio.h>
5      #include "calculate.h"
6
7      int
8      main (void)
9      {
10     float Numeral;
```

рис. 5: строки кода.

- Просмотрел с 12 по 15 строки исходного кода (рис. 6).

```
(gdb) list 12,15
12     float Result;
13     printf("Число: ");
14     scanf("%f",&Numeral);
15     printf("Операция (+,-,*,/,pow,sqrt,sin,cos,tan): ");
```

рис. 6: 12-15 строки кода.

- Просмотрел несколько строк неосновного файла (рис. 7).

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

рис. 7: строки неосновного файла.

- Установил точку останова на строке 21 (рис. 8).

```
(gdb) break 21
Breakpoint 1 at 0x40120f: file calculate.c, line 21.
```

рис. 8: создание точки останова.

- Вывел информацию о точках останова (рис. 9).

```
(gdb) info breakpoints
Num      Type           Disp Enb Address              What
1        breakpoint     keep y   0x000000000040120f in Calculate
                                     at calculate.c:21
```

рис. 9: информация о точках останова.

- Еще раз запустил программу (рис. 10).

```
(gdb) run
Starting program: /home/edsmirnovmaljce/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): -
Breakpoint 1, Calculate (Numeral=5, Operation=0x7fffffffde54 "-") at calculate.c:21
21      printf("Вычитаемое: ");
```

рис. 10: запуск программы.

- Проверил значение переменной Numeral 2 способами (рис. 11).

```
(gdb) print Numeral
$1 = 5
(gdb) display Numeral
1: Numeral = 5
```

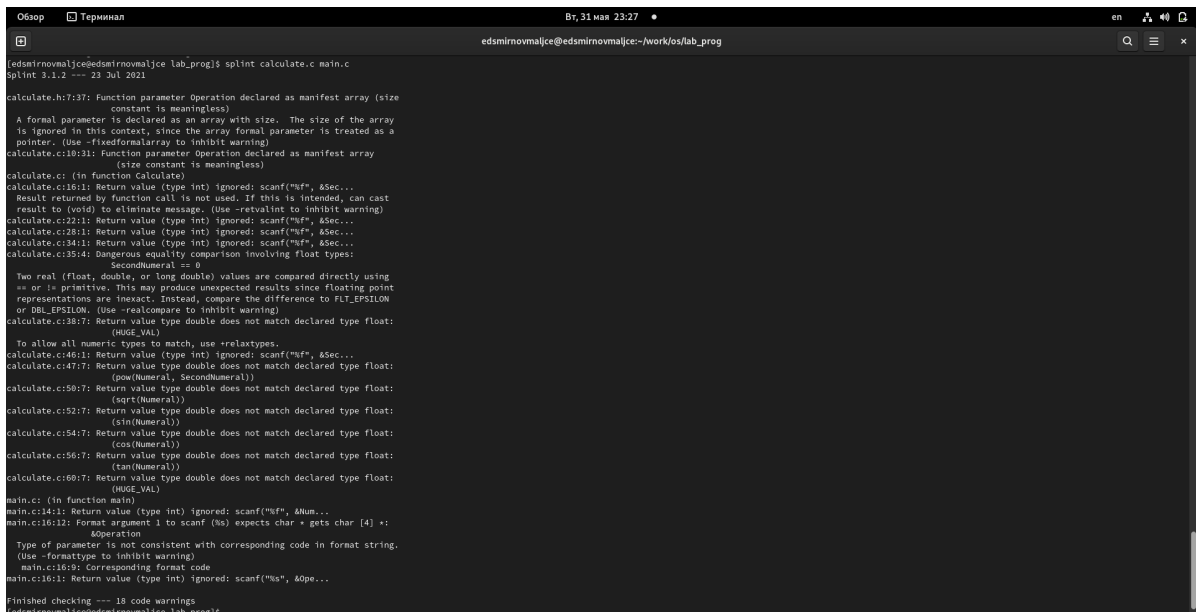
рис. 11: вывод значения переменной Numeral.

- Убрал точки останова (рис. 12).

```
(gdb) info breakpoints
Num      Type           Disp Enb Address              What
1        breakpoint     keep y   0x000000000040120f in Calculate at calculate.c:21
                                     breakpoint already hit 1 time
(gdb) delete 1
```

рис. 12: удаление точек останова.

7. С помощью splint просмотрел коды файлов main.c и calculate.c (рис. 13).



```
Обзор Терминал B7,31 май 23:27 en
edsminovmaljce@edsminovmaljce:~/work/os/lab_prog

[edsminovmaljce@edsminovmaljce lab_prog]$ splint calculate.c main.c
Splint 3.1.2 --- 23 Jul 2021

calculate.h:7:37: 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:10:31: Function parameter Operation declared as manifest array
    (size constant is meaningless)
calculate.c: (in function Calculate)
calculate.c:16:11: 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:22:11: Return value (type int) ignored: scanf("%f", &sec...
calculate.c:28:11: Return value (type int) ignored: scanf("%f", &sec...
calculate.c:34:11: Return value (type int) ignored: scanf("%f", &sec...
calculate.c:35:14: 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
    representations are inexact. Instead, compare the difference to FLT_EPSILON
    or DBL_EPSILON. (Use -realcompare to inhibit warning)
calculate.c:38:7: Return value type double does not match declared type float:
    (HUGE_VAL)
    To allow all numeric types to match, use -relaxtypes.
calculate.c:46:11: Return value (type int) ignored: scanf("%f", &sec...
calculate.c:47:7: Return value type double does not match declared type float:
    (pow(Numeral, SecondNumeral))
calculate.c:50:7: Return value type double does not match declared type float:
    (sqrt(Numeral))
calculate.c:52:7: Return value type double does not match declared type float:
    (sin(Numeral))
calculate.c:54:7: Return value type double does not match declared type float:
    (cos(Numeral))
calculate.c:56:7: Return value type double does not match declared type float:
    (tan(Numeral))
calculate.c:60:7: Return value type double does not match declared type float:
    (HUGE_VAL)
main.c: (in function main)
main.c:14:11: Return value (type int) ignored: scanf("%f", &num...
main.c:16:12: Format argument 1 to scanf (%s) expects char * gets char [4] *:
    &operation
    Type of parameter is not consistent with corresponding code in format string.
    (Use -formattype to inhibit warning)
main.c:16:9: Corresponding format code
main.c:16:11: Return value (type int) ignored: scanf("%s", &ope...

Finished checking --- 18 code warnings
[edsminovmaljce@edsminovmaljce lab_prog]$
```

рис. 13: коды файлов.

5 Ответы на контрольные вопросы

1. Дополнительную информацию о этих программах можно получить с помощью утилиты `man`.
2. Unix поддерживает следующие основные этапы разработки приложений:
 - создание исходного кода программы;
 - представление в виде файла;
 - сохранение различных вариантов исходного текста;
 - анализ исходного текста;
 - компиляция исходного текста и построение исполняемого модуля;
 - тестирование и отладка;
 - проверка кода на наличие ошибок
 - сохранение всех изменений, выполняемых при тестировании и отладке.
3. 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 уста-

новки 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. Утилита `make` освобождает пользователя от такой ручной компиляции всех файлов 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` может 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.

7. Пошаговая отладка программ заключается в том, что выполняется один оператор программы и, затем контролируются те переменные, на которые должен был воздействовать данный оператор.

8. Основные команды `gdb`:
 - `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;

9. Схема отладки программы которую мы использовали при выполнении лабораторной работы.

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. При первом запуске программы с синтаксической ошибкой отладчику не понравился формат `%s` для `&Operation`, т.к `%s` — символьный формат, а значит необходим только `Operation`.

11. При работе с исходным кодом, который не вами разрабатывался, 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. Основные задачи, решаемые программой `splint`:

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.

6 Выводы

- Я научился писать командные файлы с использованием управляющих конструкций и циклов.