

Лабораторная работа No 13.

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

Сарасбати Брасалес

Содержание

1	Цель работы	5
2	Задание	6
3	Теоретическое введение	7
4	Выполнение лабораторной работы	8
5	Выводы	12
6	Контрольные вопросы	13
	Список литературы	18

Список иллюстраций

4.1	8
4.2	8
4.3	9
4.4	9
4.5	10
4.6	10
4.7	11
4.8	11

Список таблиц

1 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.

2 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. Реализация функций калькулятора в файле `calculate.h`:
3. Выполните компиляцию программы посредством `gcc`:
`1 gcc -c calculate.c 2 gcc -c main.c 3 gcc calculate.o main.o -o calcul -lm`
4. При необходимости исправьте синтаксические ошибки.
5. Создайте `Makefile`
6. С помощью `gdb` выполните отладку программы `calcul` (перед использованием `gdb` исправьте `Makefile`)
7. С помощью утилиты `splint` попробуйте проанализировать коды файлов `calculate.c` и `main.c`.

3 Теоретическое введение

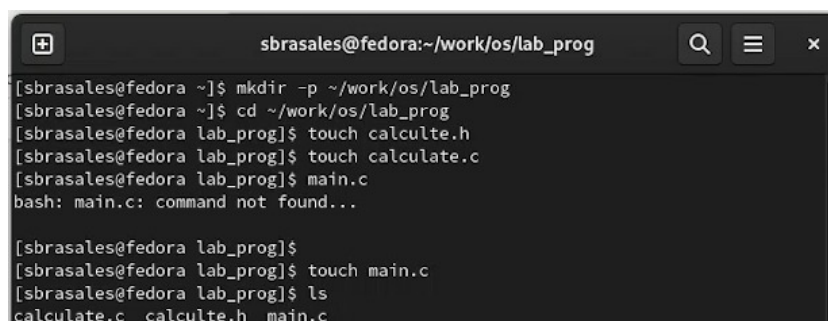
13.2.1. Этапы разработки приложений Процесс разработки программного обеспечения обычно разделяется на следующие этапы: – планирование, включающее сбор и анализ требований к функционалу и другим характеристикам разрабатываемого приложения; – проектирование, включающее в себя разработку базовых алгоритмов 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.

13.2.2. 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 gcs, которая интерпретирует аргументы командной строки, определяет и осуществляет запуск нужного компилятора для входного файла. Файлы с расширением (суффиксом) .c воспринимаются gcs как программы на языке C, файлы с расширением .cc или .C — как файлы на языке C++, а файлы с расширением .o считаются объектными.

4 Выполнение лабораторной работы

Я создала новую папку “lab_prog” и несколько файлов “calculate.h”, “calculate.c” “main.c

В файлы я ввела код лабораторного файла

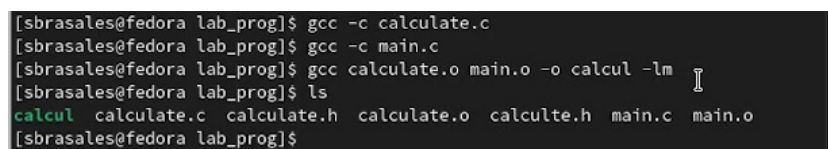


```
sbrasales@fedora:~/work/os/lab_prog
[sbrasales@fedora ~]$ mkdir -p ~/work/os/lab_prog
[sbrasales@fedora ~]$ cd ~/work/os/lab_prog
[sbrasales@fedora lab_prog]$ touch calculate.h
[sbrasales@fedora lab_prog]$ touch calculate.c
[sbrasales@fedora lab_prog]$ touch main.c
bash: main.c: command not found...

[sbrasales@fedora lab_prog]$
[sbrasales@fedora lab_prog]$ touch main.c
[sbrasales@fedora lab_prog]$ ls
calculate.c  calculate.h  main.c
```

Рис. 4.1:

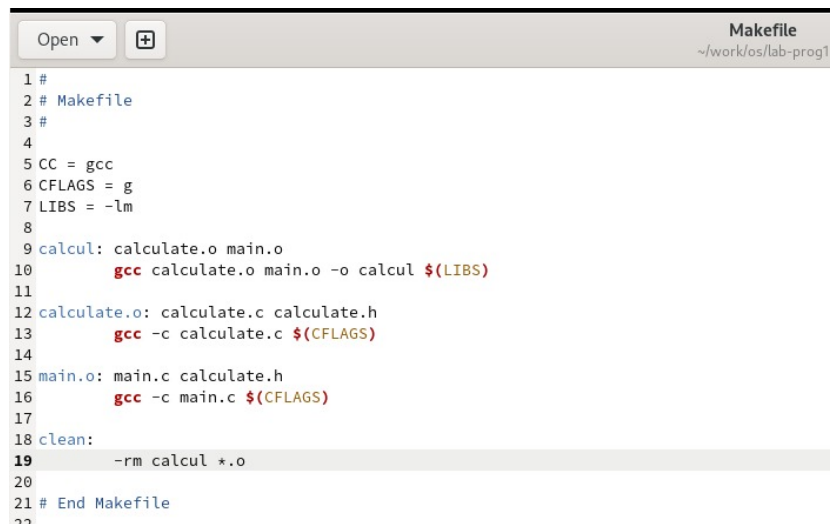
Затем выполнила компиляцию программы посредством gcc, используя команды:



```
[sbrasales@fedora lab_prog]$ gcc -c calculate.c
[sbrasales@fedora lab_prog]$ gcc -c main.c
[sbrasales@fedora lab_prog]$ gcc calculate.o main.o -o calcul -lm
[sbrasales@fedora lab_prog]$ ls
calcul  calculate.c  calculate.h  calculate.o  calculate.h  main.c  main.o
[sbrasales@fedora lab_prog]$
```

Рис. 4.2:

Я создала файл “Makefile” и исправила строки 6 и 19



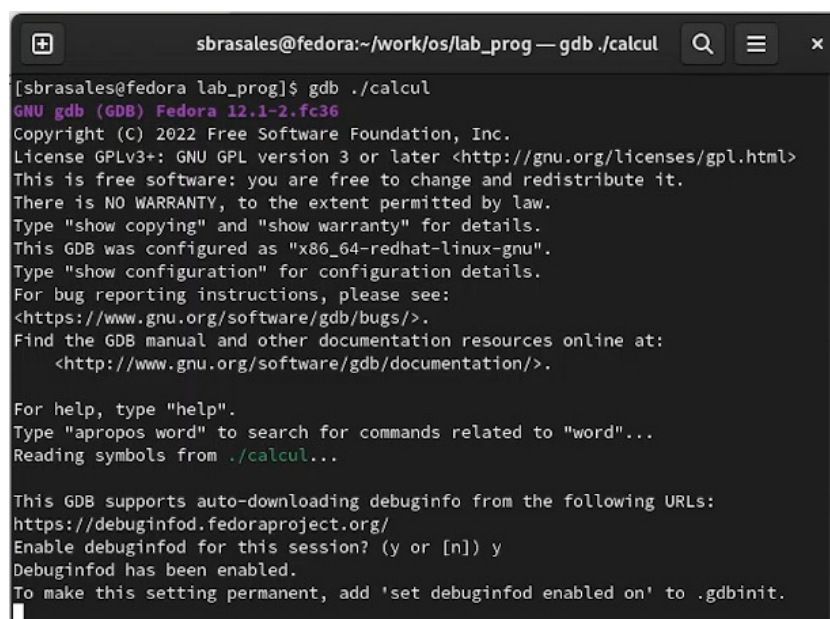
```

1 #
2 # Makefile
3 #
4
5 CC = gcc
6 CFLAGS = g
7 LIBS = -lm
8
9 calcul: calculate.o main.o
10     gcc calculate.o main.o -o calcul $(LIBS)
11
12 calculate.o: calculate.c calculate.h
13     gcc -c calculate.c $(CFLAGS)
14
15 main.o: main.c calculate.h
16     gcc -c main.c $(CFLAGS)
17
18 clean:
19     -rm calcul *.o
20
21 # End Makefile
22

```

Рис. 4.3:

После я запустила отладчик GDB командой `gdb ./calcul`, загрузив в него программу для отладки.



```

sbrasales@fedora:~/work/os/lab_prog — gdb ./calcul
[sbrasales@fedora lab_prog]$ gdb ./calcul
GNU gdb (GDB) Fedora 12.1-2.fc36
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 ./calcul...

This GDB supports auto-downloading debuginfo from the following URLs:
https://debuginfod.fedoraproject.org/
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.

```

Рис. 4.4:

Запустила программу внутри отладчика, используя команду `run`

```
sbrasales@fedora:~/work/os/lab_prog — gdb ./calcul
(gdb) run
Starting program: /home/sbrasales/work/os/lab_prog/calcul
7
Downloading 0.01 MB separate debug info for system-supplied DSO at 0x7ffff7fc400
0
Downloading 2.25 MB separate debug info for /lib64/libm.so.6
Downloading 7.42 MB 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: Операция (+,-,*,/,pow,sqrt,sin,cos,tan): +
Второе слагаемое: 2
9.00
[Inferior 1 (process 40015) exited normally]
(gdb) run
Starting program: /home/sbrasales/work/os/lab_prog/calcul
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Число: 4
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): *
Множитель: 4
16.00
[Inferior 1 (process 40028) exited normally]
(gdb)
```

Рис. 4.5:

Я использовала «list», чтобы увидеть код, а затем «list 12,15», чтобы увидеть исходный код

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

Рис. 4.6:

break,infor breakpoints

```

(gdb) break 8
Breakpoint 1 at 0x40148e: file main.c, line 13.
(gdb) info breakpoints
Num      Type             Disp Enb Address            What
1        breakpoint      keep y   0x000000000040148e in main at main.c:13
(gdb) print numeral
No symbol "numeral" in current context.
(gdb) print NUmeral
No symbol "NUmeral" in current context.
(gdb) run
Starting program: /home/sbrasales/work/os/lab-prog1/calcul
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Breakpoint 1, main () at main.c:13
13      printf("Числа: ");

```

Рис. 4.7:

splint main.c

```

sbrasales@fedora:~/work/os/lab-prog1
Finished checking --- 4 code warnings

[sbrasales@fedora lab-prog1]$ splint main.c
Splint 3.1.2 --- 22 Jan 2022

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)
main.c: (in function main)
main.c:14:1: Return value (type int) ignored: scanf("%f", &Num...
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)
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:16:1: Return value (type int) ignored: scanf("%s", &ope...

Finished checking --- 4 code warnings
[sbrasales@fedora lab-prog1]$

```

Рис. 4.8:

5 Выводы

Мы приобрели простейшие навыки разработки, анализа, тестирования и отладки приложений в ОС типа UNIX/Linux на примере создания на языке программирования С калькулятора с простейшими функциями.

6 Контрольные вопросы

1. 1. Как получить более полную информацию о программах: gcc, make, gdb и др.?

Дополнительную информацию о этих программах можно получить с помощью функций info и man. 2. Назовите и дайте краткую характеристику основным этапам разработки приложений в UNIX? 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. Что такое суффиксы и префиксы? Основное их назначение. Приведите примеры их использования. 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 в UNIX? Основное назначение компилятора с языка Си заключается в компиляции всей программы в целом и получении исполняемого модуля.

5. Для чего предназначена утилита make? 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. Приведите структуру make-файла. Дайте характеристику основным элементам этого файла. `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 стро-

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

- 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. Опишите по шагам схему отладки программы которую вы использовали при выполнении лабораторной работы. Выполнили компиляцию программы Увидели ошибки в программе (если они есть) Открыли редактор и исправили программу Загрузили программу в отладчик `gdb run` — отладчик выполнил программу, мы ввели требуемые значения. программа завершена, `gdb` не видит ошибок. 10. Прокомментируйте реакцию компилятора на синтаксические ошибки в программе при его первом запуске. Ошибок не было. 11. 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: – `cscope` - исследование функций, содержащихся в программе; – `splint` — критическая проверка программ, написанных на языке Си. 12. Каковы основные задачи, решаемые программой `slint`? • Проверка

корректности задания аргументов всех использованных в программе функций, а также типов возвращаемых ими значений; • Поиск фрагментов исходного текста, корректных с точки зрения синтаксиса языка Си, но малоэффективных с точки зрения их реализации или содержащих в себе семантические ошибки; • Общая оценка мобильности пользовательской программы.

Список литературы