

Лабораторная работа №13

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

Чемоданова Ангелина Александровна

Содержание

1	Цель работы	5
2	Задание	6
3	Теоретическое введение	8
4	Выполнение лабораторной работы	9
5	Выводы	13
6	Контрольные вопросы	14

Список иллюстраций

4.1	Каталог, файлы и компиляция	9
4.2	Запуск программы calcul	10
4.3	Работа в gdb	11
4.4	Splint	12

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.
3. Выполните компиляцию программы посредством `gcc`.
4. При необходимости исправьте синтаксические ошибки.
5. Создайте `Makefile`.
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`: `1 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. Отладчик выдаст следующую информацию: `#0 Calculate (Numeral=5, Operation=0x7ffffffd280 "-") 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

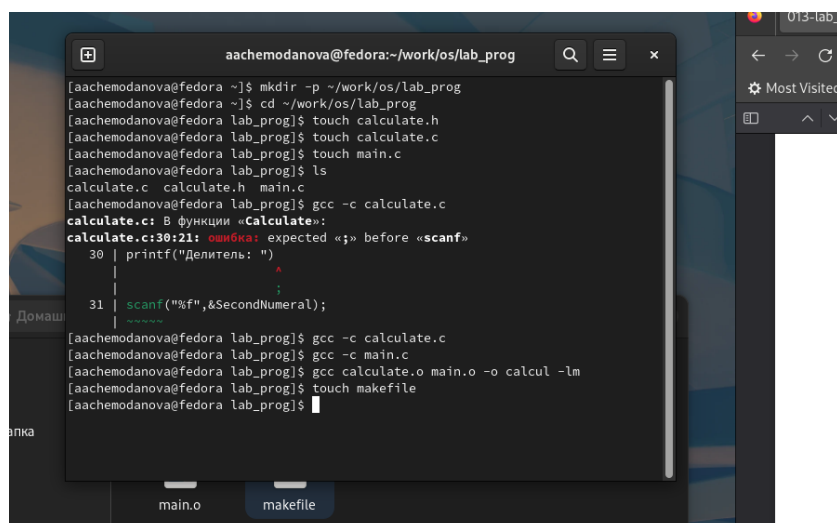
7. С помощью утилиты `splint` попробуйте проанализировать коды файлов `calculate.c` и `main.c`.

3 Теоретическое введение

Командный процессор (командная оболочка, интерпретатор команд shell) — это программа, позволяющая пользователю взаимодействовать с операционной системой компьютера. В операционных системах типа UNIX/Linux наиболее часто используются следующие реализации командных оболочек: – оболочка Борна (Bourne shell или sh) — стандартная командная оболочка UNIX/Linux, содержащая базовый, но при этом полный набор функций; – C-оболочка (или csh) — надстройка на оболочкой Борна, использующая C-подобный синтаксис команд с возможностью сохранения истории выполнения команд; – оболочка Корна (или ksh) — напоминает оболочку C, но операторы управления программой совместимы с операторами оболочки Борна; – BASH — сокращение от Bourne Again Shell (опять оболочка Борна), в основе своей совмещает свойства оболочек C и Корна (разработка компании Free Software Foundation). POSIX (Portable Operating System Interface for Computer Environments) — набор стандартов описания интерфейсов взаимодействия операционной системы и прикладных программ.

4 Выполнение лабораторной 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.
3. Выполните компиляцию программы посредством `gcc`. (рис. 4.1).

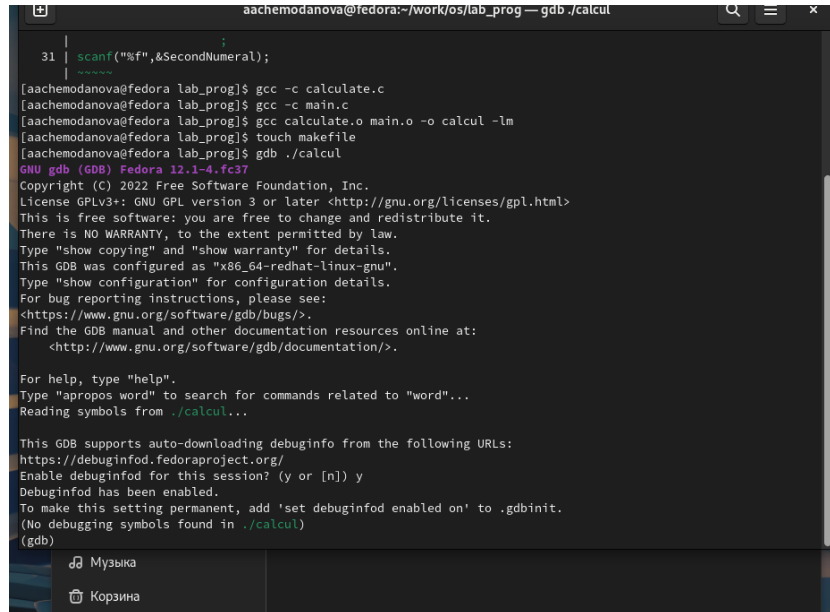


```
aachemodanova@fedora:~/work/os/lab_prog
[aachemodanova@fedora ~]$ mkdir -p ~/work/os/lab_prog
[aachemodanova@fedora ~]$ cd ~/work/os/lab_prog
[aachemodanova@fedora lab_prog]$ touch calculate.h
[aachemodanova@fedora lab_prog]$ touch calculate.c
[aachemodanova@fedora lab_prog]$ touch main.c
[aachemodanova@fedora lab_prog]$ ls
calculate.c calculate.h main.c
[aachemodanova@fedora lab_prog]$ gcc -c calculate.c
calculate.c: В функции «Calculate»:
calculate.c:30:21: ошибка: expected «$» before «scanf»
   30 | printf("Делитель: ")
      |                   ^
   31 | scanf("%f",&SecondNumeral);
      |                   ^
[aachemodanova@fedora lab_prog]$ gcc -c calculate.c
[aachemodanova@fedora lab_prog]$ gcc -c main.c
[aachemodanova@fedora lab_prog]$ gcc calculate.o main.o -o calcul -lm
[aachemodanova@fedora lab_prog]$ touch makefile
[aachemodanova@fedora lab_prog]$
```

Рис. 4.1: Каталог, файлы и компиляция

4. При необходимости исправьте синтаксические ошибки.
5. Создайте Makefile.

6. С помощью gdb выполните отладку программы calcul (перед использованием gdb исправьте Makefile): Запустите отладчик GDB, загрузив в него программу для отладки: `gdb ./calcul`. (рис. 4.2).



```
aachemodanova@fedora:~/work/os/lab_prog — gdb ./calcul
31 | scanf("%f",&SecondNumeral);
    |
[aachemodanova@fedora lab_prog]$ gcc -c calculate.c
[aachemodanova@fedora lab_prog]$ gcc -c main.c
[aachemodanova@fedora lab_prog]$ gcc calculate.o main.o -o calcul -lm
[aachemodanova@fedora lab_prog]$ touch makefile
[aachemodanova@fedora lab_prog]$ gdb ./calcul
GNU gdb (GDB) Fedora 12.1-4.fc37
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.
(No debugging symbols found in ./calcul)
(gdb)
```

Рис. 4.2: Запуск программы calcul

Для запуска программы внутри отладчика введите команду `run`: `run` Для страничного (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. Отладчик выдаст следующую информацию: `#0 Calculate (Numeral=5, Operation=0x7fffffff280 "-") at calculate.c:21 #1 0x000000000400b2b 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.(рис. 4.3).

```

aachemodanova@fedora:~/work/os/lab_prog — gdb ./calcul
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
(No debugging symbols found in ./calcul)
(gdb) run
Starting program: /home/aachemodanova/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: 1
4.00
[Inferior 1 (process 3367) exited normally]
(gdb) list
1      /* Terminate the frame unwind info section with a 4byte 0 as a sentinel;
2      this would be the 'length' field in a real FDE.  */
3
4      typedef unsigned int u32 __attribute__((mode(SI)));
5      static const u32 __FRAME_END__ = 1;
6      __attribute__((used, section(".eh_frame")))
7      0;
(gdb) list 12,15
line number 12 out of range; sofini.c has 7 lines.
(gdb) list calculate.c:20,29
No source file named calculate.c.
(gdb) list calculate.c:20,27
No source file named calculate.c.
(gdb) break 21
No line 21 in the current file.
Make breakpoint pending on future shared library load? (y or [n]) y
(gdb) info breakpoints
Num   Type             Disp Enb Address          What
1     breakpoint       keep y   <PENDING> 21
(gdb) run
Starting program: /home/aachemodanova/work/os/lab_prog/calcul
Downloading 0.00 MB source file /usr/src/debug/glibc-2.30-7.fc37.x86_64/elf/dl-call-libc-early-init.c
Breakpoint 1, _dl_call_libc_early_init (libc_map=0x7ffff7fc15d0, initial_initialentry=true) at dl-call-libc-early-init.c:27
27
(gdb) 5
Undefined command: "5". Try "help".
(gdb) print Numeral
No symbol "Numeral" in current context.
(gdb) info breakpoints
Num   Type             Disp Enb Address          What
1     breakpoint       keep y   0x00007ffff7fcad00 in _dl_call_libc_early_init at dl-call-libc-early-init.c:27
1     breakpoint already hit 1 time
(gdb) delete 1
(gdb)

```

Рис. 4.3: Работа в gdb

7. С помощью утилиты `splint` попробуйте проанализировать коды файлов `calculate.c` и `main.c`. (рис. 4.4).

```
aachemodanova@fedora:~/work/os/lab_prog
representations are inexact. Instead, compare the difference to FLT_EPSILON
or DBL_EPSILON. (Use -realcompare to inhibit warning)
calculate.c:35:7: Return value type double does not match declared type float:
(HUGE_VAL)
To allow all numeric types to match, use +relaxtypes.
calculate.c:43:1: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:44:7: Return value type double does not match declared type float:
(pow(Numeral, SecondNumeral))
calculate.c:47:7: Return value type double does not match declared type float:
(sqrt(Numeral))
calculate.c:49:7: Return value type double does not match declared type float:
(sin(Numeral))
calculate.c:51:7: Return value type double does not match declared type float:
(cos(Numeral))
calculate.c:53:7: Return value type double does not match declared type float:
(tan(Numeral))
calculate.c:57:7: Return value type double does not match declared type float:
(HUGE_VAL)

Finished checking --- 15 code warnings
[aachemodanova@fedora lab_prog]$ splint main.c
Splint 3.1.2 --- 23 Jul 2022

calculate.h:4:37: 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:10:1: 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:12:12: 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:12:9: Corresponding format code
main.c:12:1: Return value (type int) ignored: scanf("%s", &Ope...

Finished checking --- 4 code warnings
[aachemodanova@fedora lab_prog]$
```

Рис. 4.4: Splint

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. 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 ~
```

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.

7. 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. 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.
8. 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. 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.