

# **Лабораторная работа №10**

**Архитектура компьютера**

Голованова Мария Константиновна

# Содержание

<b>1</b>	<b>Цель работы</b>	<b>5</b>
<b>2</b>	<b>Задание</b>	<b>6</b>
<b>3</b>	<b>Теоретическое введение</b>	<b>7</b>
<b>4</b>	<b>Выполнение лабораторной работы</b>	<b>8</b>
4.1	Реализация подпрограмм в NASM . . . . .	8
4.2	Отладка программ с помощью GDB . . . . .	12
4.2.1	Добавление точек останова . . . . .	17
4.2.2	Работа с данными программы в GDB . . . . .	18
4.2.3	Обработка аргументов командной строки в GDB . . . . .	22
<b>5</b>	<b>Выполнение самостоятельной работы</b>	<b>25</b>
<b>6</b>	<b>Выводы</b>	<b>26</b>

## Список иллюстраций

4.1	Создание каталога для программ лабораторной работы №10 и файла lab10-1.asm . . . . .	8
4.2	Введение текста программы из листинга 10.1 . . . . .	9
4.3	Создание и проверка исполняемого файла lab10-1 . . . . .	10
4.4	Изменение текста программы в файле lab10-1.asm . . . . .	11
4.5	Изменение текста программы в файле lab10-1.asm . . . . .	12
4.6	Результат работы изменённого файла lab10-1.asm . . . . .	12
4.7	Создание файла lab10-2.asm . . . . .	12
4.8	Введение текста программы из листинга 10.2 . . . . .	13
4.9	Создание исполняемого файла lab10-2 . . . . .	14
4.10	Загрузка исполняемого файла в отладчик gdb . . . . .	14
4.11	Проверка работы программы в оболочке GDB . . . . .	14
4.12	Установка брейкпоинта на метку _start и запуск программы . . .	15
4.13	Просмотр дисассимилированного кода программы начиная с метки _start с помощью команды disassemble . . . . .	15
4.14	Переключение на отображение команд с синтаксисом Intel . . . .	16
4.15	Режим псевдографики . . . . .	17
4.16	Проверка наличия точки останова, установка новой точки останова, проверка наличия точек . . . . .	18
4.17	Выполнение 5 инструкций с помощью команды stepi . . . . .	19
4.18	Просмотр содержимого регистров с помощью команды info registers	20
4.19	Просмотр значения переменной msg1 по имени . . . . .	20
4.20	Просмотр значения переменной msg2 по адресу инструкции mov esx,msg2 . . . . .	20
4.21	Изменение первого символа переменной msg1 . . . . .	21
4.22	Изменение символа в переменной msg2 . . . . .	21
4.23	Выведение значения регистра edx в различных форматах . . . . .	21
4.24	Изменение значения регистра ebx . . . . .	22
4.25	Завершение выполнения программы и выход из GDB . . . . .	22
4.26	Копирование файла lab9-2.asm в файл с именем lab10-3.asm и создание исполняемого файла . . . . .	23
4.27	Загрузка исполняемого файла в отладчик с указанием аргументов	23
4.28	Установка точки останова перед первой инструкцией и запуск программы . . . . .	23
4.29	Просмотр позиций стека . . . . .	24

## Список таблиц

# 1 Цель работы

Приобретение навыков написания программ с использованием подпрограмм. Знакомство с методами отладки при помощи GDB и его основными возможностями.

## 2 Задание

Создать программы, вычисляющие результат с использованием подпрограмм.  
Исследовать программы с помощью GDB.

### 3 Теоретическое введение

Подпрограмма — часть компьютерной программы, содержащая описание определённого набора действий, которая может быть многократно вызвана из разных частей программы.

Отладка — процесс поиска и исправления ошибок в программе. В общем случае его можно разделить на четыре этапа: обнаружение ошибки, поиск её местонахождения, определение причины ошибки и исправление ошибки. Наиболее часто применяют такие методы отладки, как создание точек контроля значений на входе и выходе участка программы (например, вывод промежуточных значений на экран) или использование специальных программ-отладчиков. Отладчики позволяют управлять ходом выполнения программы, контролировать и изменять данные, что помогает быстрее найти место ошибки в программе и ускорить её исправление. Они позволяют увидеть, что происходит «внутри» программы в момент её выполнения или что делает программа в момент сбоя. Наиболее известным отладчиком для Linux является программа GNU GDB (GNU Debugger — отладчик проекта GNU). Он работает на многих UNIX-подобных системах и умеет производить отладку многих языков программирования. Самые популярные способы работы с отладчиком — это использование точек останова и выполнение программы по шагам.

## 4 Выполнение лабораторной работы

### 4.1 Реализация подпрограмм в NASM

Я создала каталог для выполнения лабораторной работы No 10, перешла в него и создала файл lab10-1.asm (рис. 4.1).

```
[mkgolovanova@fedora ~]$ mkdir ~/work/arch-pc/lab10  
[mkgolovanova@fedora ~]$ cd ~/work/arch-pc/lab10  
[mkgolovanova@fedora lab10]$ touch lab10-1.asm  
[mkgolovanova@fedora lab10]$
```

Рис. 4.1: Создание каталога для программ лабораторной работы №10 и файла lab10-1.asm

В качестве примера я рассмотрела программу вычисления арифметического выражения  $f(x) = 2x + 7$  с помощью подпрограммы `_calcul`. В данном примере `x` вводится с клавиатуры, а само выражение вычисляется в подпрограмме. Я внимательно изучила текст программы из листинга 10.1, ввела его в файл lab10-1.asm, создала исполняемый файл и проверила его работу (рис. 4.2, рис. 4.3).



```

GNU nano 6.0 /home/mkgolovanova/work/arc
#include 'in_out.asm'
SECTION .data
msg: DB 'Введите x: ',0
result: DB '2x+7=',0
SECTION .bss
x: RESB 80
res: RESB 80
SECTION .text
GLOBAL _start
_start:
;-----
; Основная программа
;-----
mov eax, msg
call sprint
mov ecx, x
mov edx, 80
call sread
mov eax,x
call atoi
call _calcul ; Вызов подпрограммы _calcul
mov eax,result
call sprint
mov eax,[res]
call iprintLF
call quit
;-----
; Подпрограмма вычисления
; выражения "2x+7"
_calcul:
mov ebx,2
mul ebx
add eax,7
mov [res],eax
ret ; выход из подпрограммы

```

Рис. 4.2: Введение текста программы из листинга 10.1

```
[mkgolovanova@fedora lab10]$ nasm -f elf lab10-1.asm
[mkgolovanova@fedora lab10]$ ld -m elf_i386 -o lab10-1 lab10-1.o
[mkgolovanova@fedora lab10]$ ./lab10-1
Введите x: 5
2x+7=17
[mkgolovanova@fedora lab10]$
```

Рис. 4.3: Создание и проверка исполняемого файла lab10-1

Я изменила текст программы, добавив подпрограмму `_subcalcul` в подпрограмму `_calcul`, для вычисления выражения  $f(g(x))$ , где  $x$  вводится с клавиатуры,  $f(x) = 2x + 7$ ,  $g(x) = 3x - 1$  (т.е.  $x$  передается в подпрограмму `_calcul` из нее в подпрограмму `_subcalcul`, где вычисляется выражение  $g(x)$ , результат возвращается в `_calcul` и вычисляется выражение  $f(g(x))$ , результат возвращается в основную программу для вывода результата на экран) (рис. 4.4, рис. 4.5, рис. 4.6).

```

GNU nano 6.0
#include 'in_out.asm'
SECTION .data
msg: DB 'Введите x: ',0
result: DB 'f(g(x))=2g(x)+7=',0
result1: DB 'g(x)=3x-1=',0
SECTION .bss
x: RESB 80
res: RESB 80
SECTION .text
GLOBAL _start
_start:
;-----
; Основная программа
;-----
mov eax, msg
call sprint
mov ecx, x
mov edx, 80
call sread
mov eax, x
call atoi
call _calcul ; Вызов подпрограммы _calcul
mov eax, result
call sprint
mov eax, [res]
call iprintLF
call quit
;-----
; Подпрограмма вычисления выражения "2g(x)+7"
_calcul:
call _subcalcul
mov ebx, 2
mul ebx
add eax, 7
mov [res], eax
ret ; выход из подпрограммы
_subcalcul:
mov ebx, 3
mul ebx
sub eax, 1
mov [res], eax
mov eax, result1

```

Рис. 4.4: Изменение текста программы в файле lab10-1.asm

```

mov [res],eax
ret ; выход из подпрограммы
_subcalcul:
mov ebx,3
mul ebx
sub eax,1
mov [res],eax
mov eax,result1
call sprint
mov eax,[res]
call iprintLF
ret ;

```

Рис. 4.5: Изменение текста программы в файле lab10-1.asm

```

[mkgolovanova@fedora lab10]$ nasm -f elf lab10-1.asm
[mkgolovanova@fedora lab10]$ ld -m elf_i386 -o lab10-1 lab10-1.o
[mkgolovanova@fedora lab10]$ ./lab10-1
Введите x: 5
g(x)=3x-1=14
f(g(x))=2g(x)+7=35
[mkgolovanova@fedora lab10]$

```

Рис. 4.6: Результат работы изменённого файла lab10-1.asm

## 4.2 Отладка программ с помощью GDB

Я создала файл lab10-2.asm и ввела в него текст программы из Листинга 10.2 (Программа печати сообщения Hello world!) (рис. 4.7, рис. 4.8).

```

[mkgolovanova@fedora lab10]$ touch lab10-2.asm
[mkgolovanova@fedora lab10]$

```

Рис. 4.7: Создание файла lab10-2.asm

```
GNU nano 6.0 /home/mkgolova
SECTION .data
msg1: db "Hello, ",0x0
msg1Len: equ $ - msg1
msg2: db "world!",0xa
msg2Len: equ $ - msg2
SECTION .text
global _start
_start:
mov eax, 4
mov ebx, 1
mov ecx, msg1
mov edx, msg1Len
int 0x80
mov eax, 4
mov ebx, 1
mov ecx, msg2
mov edx, msg2Len
int 0x80
mov eax, 1
mov ebx, 0
int 0x80

```

Рис. 4.8: Введение текста программы из листинга 10.2

Я получила исполняемый файл (рис. 4.9). Для работы с GDB в исполняемый файл необходимо добавить отладочную информацию, для этого трансляцию

программ необходимо проводить с ключом '-g'.

```
[mkgolovanova@fedora lab10]$ nasm -f elf lab10-2.asm
[mkgolovanova@fedora lab10]$ ld -m elf_i386 -o lab10-2 lab10-2.o
[mkgolovanova@fedora lab10]$ ./lab10-2
Hello, world!
[mkgolovanova@fedora lab10]$ nasm -f elf -g -l lab10-2.lst lab10-2.asm
[mkgolovanova@fedora lab10]$ ld -m elf_i386 -o lab10-2 lab10-2.o
[mkgolovanova@fedora lab10]$
```

Рис. 4.9: Создание исполняемого файла lab10-2

Я загрузила исполняемый файл в отладчик gdb (рис. 4.10).

```
[mkgolovanova@fedora lab10]$ gdb lab10-2
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 lab10-2...
```

Рис. 4.10: Загрузка исполняемого файла в отладчик gdb

Я проверила работу программы, запустив ее в оболочке GDB с помощью команды run (сокращённо r) (рис. 4.11).

```
(gdb) run
Starting program: /home/mkgolovanova/work/arch-pc/lab10/lab10-2
Hello, world!
[Inferior 1 (process 7622) exited normally]
(gdb)
```

Рис. 4.11: Проверка работы программы в оболочке GDB

Для более подробного анализа программы я установила брейкпоинт на метку \_start, с которой начинается выполнение любой ассемблерной программы, и запустила программу (рис. 4.12).

```
(gdb) break _start
Breakpoint 1 at 0x08049000: file lab10-2.asm, line 9.
(gdb) run
Starting program: /home/mkgolovanova/work/arch-pc/lab10/lab10-2

Breakpoint 1, _start () at lab10-2.asm:9
9      mov eax, 4
(gdb) █
```

Рис. 4.12: Установка брейкпоинта на метку `_start` и запуск программы

Я посмотрела дисассимилированный код программы с помощью команды `disassemble` начиная с метки `_start` (рис. 4.13).

```
(gdb) disassemble _start
Dump of assembler code for function _start:
=> 0x08049000 <+0>:      mov     $0x4,%eax
    0x08049005 <+5>:      mov     $0x1,%ebx
    0x0804900a <+10>:     mov     $0x804a000,%ecx
    0x0804900f <+15>:     mov     $0x8,%edx
    0x08049014 <+20>:     int     $0x80
    0x08049016 <+22>:     mov     $0x4,%eax
    0x0804901b <+27>:     mov     $0x1,%ebx
    0x08049020 <+32>:     mov     $0x804a008,%ecx
    0x08049025 <+37>:     mov     $0x7,%edx
    0x0804902a <+42>:     int     $0x80
    0x0804902c <+44>:     mov     $0x1,%eax
    0x08049031 <+49>:     mov     $0x0,%ebx
    0x08049036 <+54>:     int     $0x80
End of assembler dump.
(gdb) █
```

Рис. 4.13: Просмотр дисассимилированного кода программы начиная с метки `_start` с помощью команды `disassemble`

Я переключилась на отображение команд с синтаксисом Intel, введя команду `set disassembly-flavor intel` (рис. 4.14).

```

(gdb) set disassembly-flavor intel
(gdb) disassemble _start
Dump of assembler code for function _start:
=> 0x08049000 <+0>:      mov     eax,0x4
    0x08049005 <+5>:      mov     ebx,0x1
    0x0804900a <+10>:     mov     ecx,0x804a000
    0x0804900f <+15>:     mov     edx,0x8
    0x08049014 <+20>:     int     0x80
    0x08049016 <+22>:     mov     eax,0x4
    0x0804901b <+27>:     mov     ebx,0x1
    0x08049020 <+32>:     mov     ecx,0x804a008
    0x08049025 <+37>:     mov     edx,0x7
    0x0804902a <+42>:     int     0x80
    0x0804902c <+44>:     mov     eax,0x1
    0x08049031 <+49>:     mov     ebx,0x0
    0x08049036 <+54>:     int     0x80
End of assembler dump.
(gdb) 

```

Рис. 4.14: Переключение на отображение команд с синтаксисом Intel

При отображении синтаксиса машинных команд регистр и его значение в режиме АТТ указывается как \$,%, а в режиме Intel - ,.

Я включила режим псевдографики для более удобного анализа программы (рис. 4.15): (gdb) layout asm (gdb) layout regs



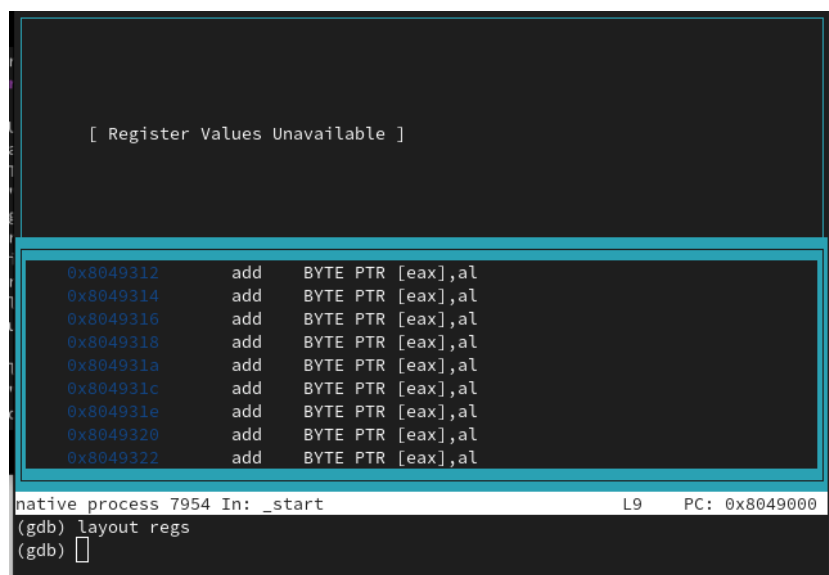
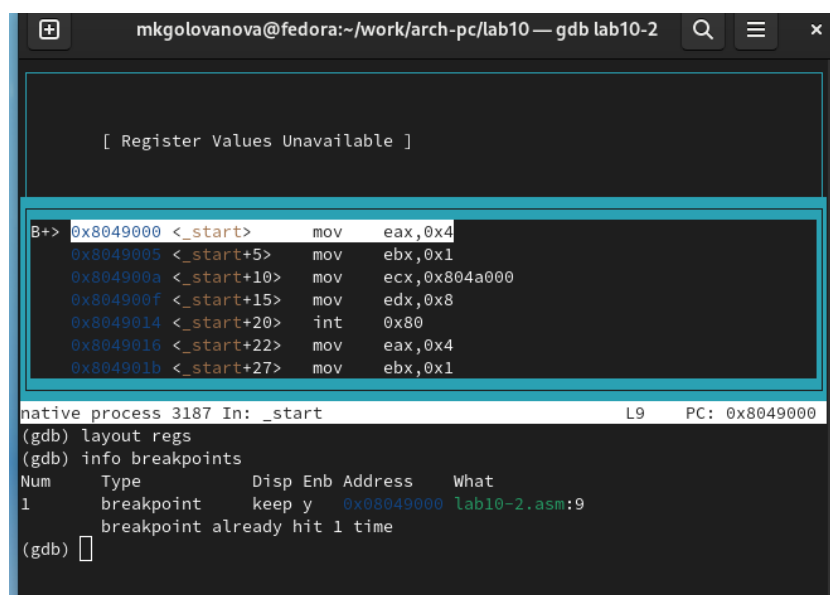


Рис. 4.15: Режим псевдографики

В этом режиме есть три окна: в верхней части видны названия регистров и их текущие значения; в средней части виден результат дисассимилирования программы; нижняя часть доступна для ввода команд.

#### 4.2.1 Добавление точек останова

Установить точку останова можно командой `break` (кратко `b`). Типичный аргумент этой команды — место установки. Его можно задать или как номер строки программы (имеет смысл, если есть исходный файл, а программа компилировалась с информацией об отладке), или как имя метки, или как адрес. Чтобы не было путаницы с номерами, перед адресом ставится «звёздочка». На предыдущих шагах была установлена точка останова по имени метки (`_start`). Я проверила это с помощью команды `info breakpoints` (кратко `i b`). Я определила адрес предпоследней инструкции (`mov ebx,0x0`), установила точку останова и посмотрела информацию о всех установленных точках останова (рис. 4.16).



The screenshot shows a GDB terminal window with the title bar "mkgolovanova@fedora:~/work/arch-pc/lab10 — gdb lab10-2". The main area displays assembly code for a native process. A blue box highlights a list of instructions. Below this, the status bar shows "native process 3187 In: \_start L9 PC: 0x8049000". The command history shows "(gdb) layout regs" and "(gdb) info breakpoints". A table of breakpoints is displayed, showing one breakpoint at address 0x8049000. The status "breakpoint already hit 1 time" is shown below the table.

```
[ Register Values Unavailable ]

B+> 0x8049000 <_start>    mov     eax,0x4
0x8049005 <_start+5>    mov     ebx,0x1
0x804900a <_start+10>   mov     ecx,0x804a000
0x804900f <_start+15>   mov     edx,0x8
0x8049014 <_start+20>   int     0x80
0x8049016 <_start+22>   mov     eax,0x4
0x804901b <_start+27>   mov     ebx,0x1

native process 3187 In: _start L9 PC: 0x8049000
(gdb) layout regs
(gdb) info breakpoints
Num   Type      Disp Enb Address      What
1     breakpoint keep y  0x08049000 lab10-2.asm:9
      breakpoint already hit 1 time
(gdb)
```

Рис. 4.16: Проверка наличия точки останова, установка новой точки останова, проверка наличия точек

## 4.2.2 Работа с данными программы в GDB

Отладчик может показывать содержимое ячеек памяти и регистров, а при необходимости позволяет вручную изменять значения регистров и переменных. Я выполнила 5 инструкций с помощью команды `stepi` (или `si`) и проследила за изменением значений регистров. Значения изменяются в регистрах `eax`, `ebx`, `ecx`, `edx`, `eax` (повторно) (рис. 4.17).

```

Register group: general
eax      0x8      8
ecx      0x804a000 134520832
edx      0x8      8
ebx      0x1      1
esp      0xffffd190 0xffffd190
ebp      0x0      0x0
esi      0x0      0
edi      0x0      0
eip      0x8049016 0x8049016 <_start+22>
eflags   0x202    [ IF ]

B+ 0x8049000 <_start>    mov    eax,0x4
   0x8049005 <_start+5>  mov    ebx,0x1
   0x804900a <_start+10> mov    ecx,0x804a000
   0x804900f <_start+15> mov    edx,0x8
   0x8049014 <_start+20> int     0x80
> 0x8049016 <_start+22> mov    eax,0x4
   0x804901b <_start+27> mov    ebx,0x1
   0x8049020 <_start+32> mov    ecx,0x804a008
   0x8049025 <_start+37> mov    edx,0x7
   0x804902a <_start+42> int     0x80

native process 7666 In: _start
      breakpoint already hit 1 time
(gdb) b *0x8049031
Breakpoint 2 at 0x8049031: file lab10-2.asm, line 20.
(gdb) i b
Num      Type           Disp Enb Address      What
1        breakpoint     keep y  0x08049000 lab10-2.asm:9
          breakpoint already hit 1 time
2        breakpoint     keep y  0x08049031 lab10-2.asm:20
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) 

```

Рис. 4.17: Выполнение 5 инструкций с помощью команды stepi

Посмотреть содержимое регистров также можно с помощью команды info registers (или i r) (рис. 4.18).

```

Register group: general
eax      0x8      8
ecx      0x804a000 134520832
edx      0x8      8
ebx      0x1      1
esp      0xffffd1b0 0xffffd1b0

0x804900a <_start+10> mov    ecx,0x804a000
0x804900f <_start+15> mov    edx,0x8
0x8049014 <_start+20> int     0x80
> 0x8049016 <_start+22> mov    eax,0x4
0x804901b <_start+27> mov    ebx,0x1
0x8049020 <_start+32> mov    ecx,0x804a008
0x8049025 <_start+37> mov    edx,0x7

native process 8372 In: _start L14 PC: 0x8049016
eax      0x8      8
ecx      0x804a000 134520832
edx      0x8      8
ebx      0x1      1
esp      0xffffd1b0 0xffffd1b0
ebp      0x0      0
esi      0x0      0
--Type <RET> for more, q to quit, c to continue without paging--

```

Рис. 4.18: Просмотр содержимого регистров с помощью команды info registers

Для отображения содержимого памяти можно использовать команду `x`, которая выдаёт содержимое ячейки памяти по указанному адресу. Формат, в котором выводятся данные, можно задать после имени команды через косую черту: `x/NFU`. С помощью команды `x &` также можно посмотреть содержимое переменной. Я посмотрела значение переменной `msg1` по имени (рис. 4.19).

```

(gdb) x/1sb &msg1
0x804a000 <msg1>:      "Hello, "
(gdb)

```

Рис. 4.19: Просмотр значения переменной `msg1` по имени

Я посмотрела значение переменной `msg2` по адресу. Адрес переменной можно определить по дизассемблированной инструкции. Я посмотрела инструкцию `mov ecx,msg2`, которая записывает в регистр `ecx` адрес переменной `msg2` (рис. 4.20)

```

0x8049000 <msg1>:      "Hello, "
(gdb) x/1sb 0x804a008
0x804a008 <msg2>:      "world!\n\034"
(gdb)

```

Рис. 4.20: Просмотр значения переменной `msg2` по адресу инструкции `mov ecx,msg2`

Изменить значение для регистра или ячейки памяти можно с помощью команды `set`, задав ей в качестве аргумента имя регистра или адрес. При этом перед именем регистра ставится префикс `$`, а перед адресом нужно указать в фигурных скобках тип данных (размер сохраняемого значения; в качестве типа данных можно использовать типы языка Си). Я изменила первый символ переменной `msg1` (рис. 4.21)

```
msg1 has unknown type, cast it to its declared type
(gdb) set {char}&msg1='h'
(gdb) x/1sb &msg1
0x804a000 <msg1>:      "hello, "
(gdb) □
```

Рис. 4.21: Изменение первого символа переменной `msg1`

Я заменила первый символ во второй переменной `msg2` (рис. 4.22)

```
0x804a000 <msg1>:      "hello, "
(gdb) set {char}&msg2='F'
(gdb) x/1bb &msg2
0x804a008 <msg2>:      "Forld!\n\034"
(gdb) □
```

Рис. 4.22: Изменение символа в переменной `msg2`

Я вывела в различных форматах (в шестнадцатеричном формате, в двоичном формате и в символьном виде) значение регистра `edx` (рис. 4.23).

```
(gdb) p/x $edx
$5 = 0x0
(gdb) p/f $edx
$6 = 0
(gdb) p/s $edx
$7 = 0
(gdb) □
```

Рис. 4.23: Выведение значение регистра `edx` в различных форматах

С помощью команды `set` я изменила значение регистра `ebx` (рис. 4.24).

```

native process 3161 In: _start
(gdb) set $ebx='2'
(gdb) p/s $ebx
$10 = 50
(gdb) set $ebx=2
(gdb) p/s $ebx
$11 = 2
(gdb) 

```

Рис. 4.24: Изменение значения регистра ebx

Разницу вывода команд `p/s $ebx` вызвана тем, что в первом случае значение регистра вводится как символ, а во втором - как цифра. Я завершила выполнение программы с помощью команды `continue` (сокращенно `c`) и вышла из GDB с помощью команды `quit` (сокращенно `q`)(рис. 4.25).

```

(gdb) c
Continuing.
hello, Forld!
Breakpoint 2, _start () at lab10-2.asm:20
(gdb) q
A debugging session is active.

        Inferior 1 [process 3161] will be killed.

Quit anyway? (y or n) y

```

Рис. 4.25: Завершение выполнения программы и выход из GDB

### 4.2.3 Обработка аргументов командной строки в GDB

Я скопировала файл `lab9-2.asm`, созданный при выполнении лабораторной работы No9, с программой, выводящей на экран аргументы командной строки (Листинг 9.2), в файл с именем `lab10-3.asm` и создала исполняемый файл (рис. 4.26).

```
[mkgolovanova@fedora lab10]$ cp ~/work/arch-pc/lab09/lab9-2.asm ~/work/arch-pc/lab10/lab10-3.asm
[mkgolovanova@fedora lab10]$ nasm -f elf -g -l lab10-3.lst lab10-3.asm
[mkgolovanova@fedora lab10]$ ld -m elf_i386 -o lab10-3 lab10-3.o
[mkgolovanova@fedora lab10]$
```

Рис. 4.26: Копирование файла lab9-2.asm в файл с именем lab10-3.asm и создание исполняемого файла

Для загрузки в gdb программы с аргументами необходимо использовать ключ `-args`. Загрузите исполняемый файл в отладчик, указав аргументы: `gdb -args lab10-3 аргумент1 аргумент 2 'аргумент 3'` (рис. 4.27).

```
[mkgolovanova@fedora lab10]$ gdb --args lab10-3 аргумент1 аргумент 2 'аргумент 3'
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 lab10-3...
(gdb)
```

Рис. 4.27: Загрузка исполняемого файл в отладчик с указанием аргументов

Как отмечалось в предыдущей лабораторной работе, при запуске программы аргументы командной строки загружаются в стек. Я исследовала расположение аргументов командной строки в стеке после запуска программы с помощью gdb. Я установила точку останова перед первой инструкцией в программе и запустила её (рис. 4.28).

```
(gdb) b _start
Note: breakpoint 1 also set at pc 0x80490e8.
Breakpoint 2 at 0x80490e8: file lab10-3.asm, line 8.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/mkgolovanova/work/arch-pc/lab10/lab10-3 аргумент1 аргумент 2 аргумент\ 3

Breakpoint 1, _start () at lab10-3.asm:8
8      pop ecx ; Извлекаем из стека в 'ecx' количество
(gdb)
```

Рис. 4.28: Установка точки останова перед первой инструкцией и запуск программы

Адрес вершины стека храниться в регистре `esp`, и по этому адресу распола-

гается число, равное количеству аргументов командной строки (включая имя программы). Число аргументов равно 5 – это имя программы lab10-3 и непосредственно аргументы: аргумент1, аргумент, 2 и 'аргумент 3'. Я посмотрела остальные позиции стека – по адресу [esp+4] располагается адрес в памяти где находится имя программы, по адресу [esp+8] храниться адрес первого аргумента, по адресу [esp+12] – второго и т.д. (рис. 4.29).

```
(gdb) x/x $esp
0xffffd140: 0x00000005
(gdb) x/s *(void**)(esp + 4)
0xffffd2fe: "/home/mkgolovanova/work/arch-pc/lab10/lab10-3"
(gdb) x/s *(void**)(esp + 8)
0xffffd32c: "аргумент1"
(gdb) x/s *(void**)(esp + 12)
0xffffd33e: "аргумент"
(gdb) x/s *(void**)(esp + 16)
0xffffd34f: "2"
(gdb) x/s *(void**)(esp + 20)
0xffffd351: "аргумент 3"
(gdb) x/s *(void**)(esp + 24)
0x0: <error: Cannot access memory at address 0x0>
(gdb) □
```

Рис. 4.29: Просмотр позиций стека

Шаг изменения адреса равен 4 ([esp+4], [esp+8], [esp+12] и т.д.), потому что количество команд для вывода каждого аргумента равно 4.



## 5 Выполнение самостоятельной работы

Я преобразовала программу из лабораторной работы №9 (Задание №1 для самостоятельной работы), реализовав вычисление значения функции  $f(x)$  как подпрограмму.

В листинге 10.3 приведена программа вычисления выражения  $(3 + 2) * 4 + 5$ . При запуске данная программа дает неверный результат. Проверьте это. С помощью отладчика GDB, анализируя изменения значений регистров, определите ошибку и исправьте ее.

## 6 Выводы

Я приобрела навыки написания программ с использованием подпрограмм и познакомилась с методами отладки при помощи GDB и его основными возможностями.