

Отчёт по лабораторной работе №9

дисциплина: Архитектура компьютера

Аносов Даниил Игоревич

Содержание

1	Цель работы	5
2	Задание	6
3	Выполнение лабораторной работы	7
3.1	Реализация подпрограмм в NASM	7
3.2	Отладка программ с помощью GDB	10
3.2.1	Добавление точек останова	14
3.2.2	Работа с данными программы в GDB	15
3.2.3	Обработка аргументов командной строки в GDB	18
4	Задание для самостоятельной работы	21
4.1	Первое задание	21
4.2	Второе задание	23
5	Выводы	27

Список иллюстраций

3.1	Создание каталога для программ	7
3.2	Открытый Vim	8
3.3	Компиляция и первый запуск программы	8
3.4	Vim с обновленной программой	9
3.5	Повторная компиляция и запуск программы	10
3.6	Vim с новой программой	10
3.7	Компиляция и запуск новой программы	11
3.8	Открытый GDB	12
3.9	Открытый GDB	13
3.10	Режим псевдографики	14
3.11	Работа в GDB	15
3.12	Работа в GDB	17
3.13	Работа в GDB	18
3.14	Открытый терминал	19
3.15	Открытый GDB	19
4.1	Vim с файлом <i>task.asm</i>	22
4.2	Проверка работы <i>task.asm</i>	23
4.3	GDB с <i>fix.asm</i>	24
4.4	GDB с <i>fix.asm</i>	25
4.5	Тестирование исправленной программы <i>fix.asm</i>	26
4.6	Загрузка файлов на GitHub	26

Список таблиц

1 Цель работы

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

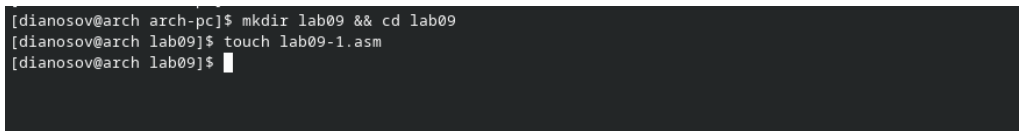
2 Задание

1. Преобразуйте программу из лабораторной работы №8 (Задание №1 для самостоятельной работы), реализовав вычисление значения функции $f(x)$ как подпрограмму.
2. В листинге 9.3 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.

3 Выполнение лабораторной работы

3.1 Реализация подпрограмм в NASM

Откроем терминал и создадим каталог для программ лабораторной работы №9. В новом каталоге создадим файл для первой программы *lab9-1.asm*. (рис. 3.1).



```
[dianosov@arch arch-pc]$ mkdir lab09 && cd lab09  
[dianosov@arch lab09]$ touch lab09-1.asm  
[dianosov@arch lab09]$
```

Рис. 3.1: Создание каталога для программ

Введём в этот файл текст программы из предложенного листинга. (рис. 3.2).

```

#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. 3.2: Открытый Vim

Первые строки программы отвечают за вывод сообщения на экран (call sprint), чтение данных введенных с клавиатуры (call sread) и преобразования введенных данных из символьного вида в численный (call atoi).

Скомпилируем и запустим программу, предварительно скопировав из каталога предыдущей лабораторной работы вспомогательный файл с подпрограммами *in_out.asm* (рис. 3.3).

```

[dianosov@arch lab09]$ cp ../lab08/in_out.asm .
[dianosov@arch lab09]$ ls
in_out.asm  lab09-1  lab09-1.asm  lab09-1.o
[dianosov@arch lab09]$ nasm -f elf lab09-1.asm
[dianosov@arch lab09]$ ld -m elf_i386 lab09-1.o -o lab09-1
[dianosov@arch lab09]$ ./lab09-1
Введите x: 1
2x+7=9
[dianosov@arch lab09]$ ./lab09-1
Введите x: 5
2x+7=17
[dianosov@arch lab09]$

```

Рис. 3.3: Компиляция и первый запуск программы

Изменим 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. 3.4).

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

Рис. 3.4: Vim с обновленной программой

Скомпилируем и запустим измененную программу. Проверим её работу. (рис. 3.5).

```
[dianosov@arch lab09]$ nasm -f elf lab09-1.asm
[dianosov@arch lab09]$ ld -m elf_i386 lab09-1.o -o lab09-1
[dianosov@arch lab09]$ ./lab09-1
Введите x: 5
2(3x-1)+7=35
[dianosov@arch lab09]$ ./lab09-1
Введите x: 1
2(3x-1)+7=11
[dianosov@arch lab09]$
```

Рис. 3.5: Повторная компиляция и запуск программы

Теперь, после того, как мы добавили новую подпрограмму `_subcalcul`, программа корректно выполняет свою задачу.

Создадим новый файл `lab09-2.asm`. Введём в него код из предложенного листинга.

Откроем файл программы в **Vim** и отредактируем файл (рис. 3.6).

```
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
```

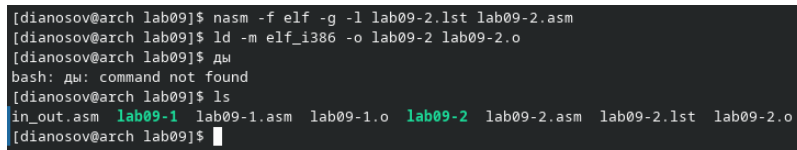
Рис. 3.6: Vim с новой программой

Проведём компиляцию новой программы, получим исполняемый файл (рис. 3.7).

3.2 Отладка программ с помощью GDB

Для работы с GDB в исполняемый файл необходимо добавить отладочную информацию, для этого трансляцию программ необходимо проводить с ключом `‘-g’`:

```
nasm -f elf -g -l lab09-2.lst lab09-2.asm
ld -m elf_i386 -o lab09-2 lab09-2.o
```



```
[dianosov@arch lab09]$ nasm -f elf -g -l lab09-2.lst lab09-2.asm
[dianosov@arch lab09]$ ld -m elf_i386 -o lab09-2 lab09-2.o
[dianosov@arch lab09]$ ды
bash: ды: command not found
[dianosov@arch lab09]$ ls
in_out.asm lab09-1 lab09-1.asm lab09-1.o lab09-2 lab09-2.asm lab09-2.lst lab09-2.o
[dianosov@arch lab09]$
```

Рис. 3.7: Компиляция и запуск новой программы

Откроем отладчик **GDB** (рис. 3.8). В нём исследуем, как работают брейкпоинты (точки останова).

```

[dianosov@arch lab09]$ gdb lab09-2
GNU gdb (GDB) 15.2
Copyright (C) 2024 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-pc-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 lab09-2...
(gdb) run
Starting program: /home/dianosov/study/arch-pc/lab09/lab09-2

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.archlinux.org>
Enable debuginfod for this session? (y or [n])
Debuginfod has been disabled.
To make this setting permanent, add 'set debuginfod enabled off' to .gdbinit.
Hello, world!
[Inferior 1 (process 8104) exited normally]
(gdb) break _start
Breakpoint 1 at 0x8049000: file lab09-2.asm, line 9.
(gdb) run
Starting program: /home/dianosov/study/arch-pc/lab09/lab09-2

Breakpoint 1, _start () at lab09-2.asm:9
9      mov eax, 4
(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) set disassembly-flavor intel
(gdb) disassemble _start
Dump of assembler code for function _start:

```

Рис. 3.8: Открытый GDB

Переключимся на отображение команд с Intel’овским синтаксисом, введя команду `set disassembly-flavor intel` (рис. 3.9). Этот режим отличается от режима *ATT* порядком операндов и стилем их обозначений, а именно, в *ATT* перед именами регистров стоят \$. Порядок операндов: в *ATT* “source, destination”, а в *Intel* “destination, source”.

```

To make this setting permanent, add 'set debuginfod enabled off' to .gdbinit.
Hello, world!
[Inferior 1 (process 8104) exited normally]
(gdb) break _start
Breakpoint 1 at 0x8049000: file lab09-2.asm, line 9.
(gdb) run
Starting program: /home/dianosov/study/arch-pc/lab09/lab09-2

Breakpoint 1, _start () at lab09-2.asm:9
9      mov eax, 4
(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) 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) layout asm

Fatal signal: Aborted
----- Backtrace -----
0x60b90a09003e ???
0x60b90a1b77c0 ???
0x79c732b5d1cf ???
0x79c732bb63f4 ???
0x79c732b5d11f ???

```

Рис. 3.9: Открытый GDB

Теперь откроем режим псевдографики для более удобного анализа программы (рис. 3.10).

```

Registers
[ Register Values Unavailable ]

b+ 0x8049000 <_start>    mov     $0x4,%eax
0x8049005 <_start+5>    mov     $0x1,%ebx
0x804900a <_start+10>   mov     $0x804a000,%ecx
0x804900f <_start+15>   mov     $0x8,%edx
0x8049014 <_start+20>   int     $0x80
0x8049016 <_start+22>   mov     $0x4,%eax
0x804901b <_start+27>   mov     $0x1,%ebx
0x8049020 <_start+32>   mov     $0x804a008,%ecx
0x8049025 <_start+37>   mov     $0x7,%edx
0x804902a <_start+42>   int     $0x80
0x804902c <_start+44>   mov     $0x1,%eax
0x8049031 <_start+49>   mov     $0x0,%ebx
0x8049036 <_start+54>   int     $0x80

exec No process (asm) In:                               L??  PC: ??
(gdb) layout regs
(gdb)

```

Рис. 3.10: Режим псевдографики

3.2.1 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`). Проверим это с помощью команды `info breakpoints` (кратко `i b`). Установим еще одну точку останова по адресу инструкции. Адрес инструкции можно увидеть в средней части экрана в левом столбце соответствующей инструкции (см. рис. 9.3). Определим адрес предпоследней инструкции (`mov ebx, 0x0`) и установим точку останова (рис. 3.11).

```

Register group: general
eax    0x0      0      ecx    0x0      0
edx    0x0      0      ebx    0x0      0
esp    0xffffd700 0xffffd700  ebp    0x0      0x0
esi    0x0      0      edi    0x0      0
eip    0x8049000 0x8049000 <_start>  eflags 0x202    [ IF ]
cs     0x23     35     ss     0x2b     43
ds     0x2b     43     es     0x2b     43
fs     0x0      0      gs     0x0      0

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
0x804902c <_start+44> mov    eax,0x1
b+ 0x8049031 <_start+49> mov    ebx,0x0
0x8049036 <_start+54> int    0x80
0x8049038 add    BYTE PTR [eax],al
0x804903a add    BYTE PTR [eax],al
0x804903c add    BYTE PTR [eax],al

native process 10439 (asm) In: _start L9 PC: 0x8049000
Starting program: /home/dianosov/study/arch-pc/lab09/lab09-2

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

Breakpoint 1, _start () at lab09-2.asm:9
(gdb) break *0x8049031
Breakpoint 3 at 0x8049031: file lab09-2.asm, line 20.
(gdb) i b
Num   Type             Disp Enb Address      What
1     breakpoint       keep y  0x08049000 lab09-2.asm:9
      breakpoint already hit 1 time
2     breakpoint       keep y  <PENDING> 0x8049031
3     breakpoint       keep y  0x08049031 lab09-2.asm:20
(gdb)

```

Рис. 3.11: Работа в GDB

3.2.2 Работа с данными программы в GDB

Отладчик может показывать содержимое ячеек памяти и регистров, а при необходимости позволяет вручную изменять значения регистров и переменных. Выполним 5 инструкций с помощью команды `stepi` (или `si`) и проследим за изменением значений регистров. Значения каких регистров изменяются? Посмотреть содержимое регистров также можно с помощью команды `info registers` (или `i r`).

```
(gdb) info registers
```

Для отображения содержимого памяти можно использовать команду `x <адрес>`, которая выдаёт содержимое ячейки памяти по указанному адресу. Формат, в котором выводятся данные, можно задать после имени команды через косую черту: `x/NFU <адрес>`. С помощью команды `x &<имя переменной>` также можно посмотреть содержимое переменной. Посмотрите значение переменной `msg1` по имени

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

Посмотрите значение переменной `msg2` по адресу. Адрес переменной можно определить по дизассемблированной инструкции. Посмотрим инструкцию `mov esx,msg2` которая записывает в регистр `esx` адрес переменной `msg2`. (рис. 3.12, 3.13).


```
Register group: general
eax      0x8      8      ecx      0x804a000      134520832
edx      0x8      8      ebx      0x1      1
esp      0xffffd700      0xffffd700      ebp      0x0      0x0
esi      0x0      0      edi      0x0      0
eip      0x8049016      0x8049016 <_start+22>      eflags      0x202      [ IF ]
cs       0x23      35      ss       0x2b      43
ds       0x2b      43      es       0x2b      43
fs       0x0      0      gs       0x0      0

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
0x804902c <_start+44>     mov     eax,0x1
b+ 0x8049031 <_start+49>     mov     ebx,0x0
0x8049036 <_start+54>     int     0x80
0x8049038      add     BYTE PTR [eax],al
0x804903a      add     BYTE PTR [eax],al
0x804903c      add     BYTE PTR [eax],al

native process 10439 (asm) In: _start      L14      PC: 0x8049016
esp      0xffffd700      0xffffd700
ebp      0x0      0x0
esi      0x0      0
edi      0x0      0
eip      0x8049000      0x8049000 <_start>
eflags   0x202      [ IF ]
cs       0x23      35
ss       0x2b      43
ds       0x2b      43
es       0x2b      43
fs       0x0      0
gs       0x0      0
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) █
```

Рис. 3.12: Работа в GDB

```

Register group: general
eax      0x8      8      ecx      0x804a000      134520832
edx      0x8      8      ebx      0x1      1
esp      0xffffd700      0xffffd700      ebp      0x0      0x0
esi      0x0      0      edi      0x0      0
eip      0x8049016      0x8049016 <_start+22>      eflags      0x202      [ IF ]
cs       0x23      35      ss       0x2b      43
ds       0x2b      43      es       0x2b      43
fs       0x0      0      gs       0x0      0

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
0x804902c <_start+44>     mov     eax,0x1
b+ 0x8049031 <_start+49>     mov     ebx,0x0
0x8049036 <_start+54>     int     0x80
0x8049038      add     BYTE PTR [eax],al
0x804903a      add     BYTE PTR [eax],al
0x804903c      add     BYTE PTR [eax],al

native process 10439 (asm) In: _start      L14      PC: 0x8049016
ecx      0x804a000      134520832
edx      0x8      8
ebx      0x1      1
esp      0xffffd700      0xffffd700
ebp      0x0      0x0
esi      0x0      0
edi      0x0      0
eip      0x8049016      0x8049016 <_start+22>
eflags   0x202      [ IF ]
cs       0x23      35
ss       0x2b      43
ds       0x2b      43
es       0x2b      43
fs       0x0      0
gs       0x0      0
(gdb) x/1sb &msg1
0x804a000 <msg1>:      "Hello, "
(gdb)

```

Рис. 3.13: Работа в GDB

3.2.3 Обработка аргументов командной строки в GDB

Скопируем файл lab8-2.asm, созданный при выполнении лабораторной работы №8, с программой выводящей на экран аргументы командной строки в файл с именем lab09-3.asm, создадим исполняемый файл (рис. 3.14).

```
[dianosov@arch lab09]$ ls ..
CHANGELOG.md                                lab05 lab09                                Makefile      README.en.md  README.tex
config                                       lab06 lab09-1.asm  meow.txt      README.git-flow.md template
COURSE                                       lab07 labs      prepare      README.md     update.sh
HI_QiYsKILxRpg3hIP6sJ7fM7Pq10NvZlMIXxw.woff2 lab08 LICENSE presentation README.pdf

[dianosov@arch lab09]$ cp ../lab08/lab8-2
lab8-2 lab8-2.asm lab8-2.o
[dianosov@arch lab09]$ cp ../lab08/lab8-2.asm lab09-3.asm
[dianosov@arch lab09]$ ls
in_out.asm lab09-1 lab09-1.o lab09-2 lab09-2.asm lab09-2.lst lab09-2.o lab09-3.asm
[dianosov@arch lab09]$ nasm -f elf -g -l lab09-3.lst lab09-3.asm
[dianosov@arch lab09]$ ld -m elf_i386 -o lab09-3 lab09-3.o
[dianosov@arch lab09]$ dy
bash: dy: command not found
[dianosov@arch lab09]$ ls
in_out.asm lab09-1.asm lab09-2 lab09-2.lst lab09-3 lab09-3.lst
lab09-1 lab09-1.o lab09-2.asm lab09-2.o lab09-3.asm lab09-3.o
[dianosov@arch lab09]$
```

Рис. 3.14: Открытый терминал

Для загрузки в gdb программы с аргументами необходимо использовать ключ `--args`. Загрузим исполняемый файл в отладчик, указав аргументы (рис. 3.15):

`gdb --args lab09-3 аргумент1 аргумент 2 'аргумент 3'`

```
[dianosov@arch lab09]$ gdb --args lab09-3 аргумент1 аргумент 2 'аргумент 3'
GNU gdb (GDB) 15.2
Copyright (C) 2024 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-pc-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 lab09-3...
(gdb) b _start
Breakpoint 1 at 0x80490e8: file lab09-3.asm, line 5.
(gdb) run
Starting program: /home/dianosov/study/arch-pc/lab09/lab09-3 аргумент1 аргумент 2 аргумент\ 3

This GDB supports auto-downloading debuginfo from the following URLs:
<https://debuginfod.archlinux.org>
Enable debuginfod for this session? (y or [n]) x/x $esp
Please answer y or [n].
Enable debuginfod for this session? (y or [n])
Debuginfod has been disabled.
To make this setting permanent, add 'set debuginfod enabled off' to .gdbinit.

Breakpoint 1, _start () at lab09-3.asm:5
5      pop ecx ; Извлекаем из стека в 'ecx' количество
(gdb) x/s *(void**)($esp + 4)
0xfffffd8af: "/home/dianosov/study/arch-pc/lab09/lab09-3"
(gdb) x/s *(void**)($esp + 8)
0xfffffd8da: "аргумент1"
(gdb) x/s *(void**)($esp + 12)
0xfffffd8ec: "аргумент"
(gdb) x/s *(void**)($esp + 16)
0xfffffd8fd: "2"
(gdb) x/s *(void**)($esp + 20)
0xfffffd8ff: "аргумент 3"
(gdb) x/s *(void**)($esp + 24)
0x0: <error: Cannot access memory at address 0x0>
(gdb)
```

Рис. 3.15: Открытый GDB

Как видно, число аргументов равно 5 – это имя программы lab09-3 и непосредственно аргументы: аргумент1, аргумент, 2 и ‘аргумент 3’. Посмотрим остальные позиции стека – по адресу [esp+4] располагается адрес в памяти где находится имя программы, по адресу [esp+8] хранится адрес первого аргумента, по адресу [esp+12] – второго и т.д.

Аргументы расположены в памяти с шагом 4 потому что в программе под каждый из них выделено по 4 байта.

4 Задание для самостоятельной работы

4.1 Первое задание

Скопируем из каталога последней лабораторной работы файл *task.asm*.

Откроем его в редакторе **Vim** (рис. 4.1).

```

#include 'in_out.asm'
SECTION .data
msg db "Результат: ",0
msg1 db "Функция: f(x)=4x-3",0
SECTION .text
global _start
_start:
mov eax, msg1
call sprintf

pop ecx ; Извлекаем из стека в 'ecx' количество
; аргументов (первое значение в стеке)
pop edx ; Извлекаем из стека в 'edx' имя программы
; (второе значение в стеке)
sub ecx,1 ; Уменьшаем 'ecx' на 1 (количество
; аргументов без названия программы)
mov esi,0 ; Используем 'esi' для хранения
; промежуточных сумм

next:
cmp ecx,0h ; проверяем, есть ли еще аргументы
jz _end ; если аргументов нет выходим из цикла
; (переход на метку '_end')
pop eax ; eax := следующий аргумент
call atoi

; имеем аргумент x в eax
; требуется сделать из него 4x-3
call _calc

add esi, eax ; esi := esi + 4x-3
mov eax, esi

loop next ; переход к обработке следующего аргумента

_end:
mov eax, msg ; вывод сообщения "Результат: "
call sprintf
mov eax, esi ; записываем сумму в регистр 'eax'
call iprintf ; печать результата
call quit ; завершение программы

_calc: ; имеем в eax значение x
mov ebx, 4
mul ebx ; eax = 4x
sub eax, 3 ; eax = 4x-3
ret ; выходим из подпрограммы
~
~
~
"task.asm" 47L, 1618B

```

Рис. 4.1: **Vim** с файлом *task.asm*

Реализуем с помощью подпрограммы вычисление значения выражения $f(x) = 4x - 3$. Для этого добавим подпрограмму `_calc`, где, имея в регистре `eax` значение переменной x , преобразуем его. Затем инструкцией `ret` выйдем из подпрограммы, оставив результат в `eax`.

Протестируем работу программы (рис. 4.2).

```

[dianosov@arch lab09]$ nasm -f elf task.asm
[dianosov@arch lab09]$ ld -m elf_i386 -o task task.o
[dianosov@arch lab09]$ ./task
Функция: f(x)=4x-3
Результат: 0
[dianosov@arch lab09]$ ./task 2
Функция: f(x)=4x-3
Результат: 5
[dianosov@arch lab09]$ ./task 2 45
Функция: f(x)=4x-3
Результат: 182
[dianosov@arch lab09]$

```

Рис. 4.2: Проверка работы *task.asm*

Выводится верный результат.

4.2 Второе задание

В листинге предложен код программы, в котором надо найти и исправить ошибку.

```

#include 'in_out.asm'
SECTION .data
div: DB 'Результат: ',0
SECTION .text
GLOBAL _start
_start:
; ---- Вычисление выражения (3+2)*4+5
mov ebx,3
mov eax,2
add ebx,eax
mov ecx,4
mul ecx
add ebx,5
mov edi,ebx
; ---- Вывод результата на экран
mov eax,div

```

```

call sprint
mov eax,edi
call iprintLF
call quit

```

Будем использовать для этого отладчик **GDB** (рис. 4.3). Установим точку останова на метке `_start`, войдём в режим отображения значений регистров, поменяем стиль имён регистров на `intel`.

The screenshot shows the GDB interface with two main panels. The top panel displays the 'Register group: general' with values for registers: eax (0x0), edx (0x0), esp (0xffffd720), esi (0x0), eip (0x80490e8), cs (0x23), ds (0x2b), fs (0x0), ecx (0x0), ebx (0x0), ebp (0x0), edi (0x0), eflags (0x202), ss (0x2b), es (0x2b), and gs (0x0). The bottom panel shows the assembly code for `fix.asm` starting at address `0x80490e8`. The code includes instructions like `mov ebx,0x3`, `mov eax,0x2`, `add ebx,eax`, `mov ecx,0x4`, `mul ecx`, `add ebx,0x5`, `mov edi,ebx`, `mov eax,0x804a000`, `call 0x804900f <sprint>`, `mov eax,edi`, `call 0x8049086 <iprintLF>`, `call 0x80490db <quit>`, and three `add BYTE PTR [eax],al` instructions. The status bar at the bottom indicates 'native process 33249 (asm) In: _start' and 'L8 PC: 0x80490e8'.

Рис. 4.3: **GDB** с *fix.asm*

Напомним: программа должна вычислять значение выражения $(3 + 2) \times 4 + 5$.
 Пройдём последовательно на несколько инструкций вперёд повторным вводом

команды `si` (рис. 4.4).

```
Register group: general
eax      0x8      8      ecx      0x4      4
edx      0x0      0      ebx      0x5      5
esp      0xffffd720 0xffffd720  ebp      0x0      0x0
esi      0x0      0      edi      0x0      0
eip      0x80490fb 0x80490fb <_start+19>  eflags   0x202    [ IF ]
cs       0x23     35     ss       0x2b     43
ds       0x2b     43     es       0x2b     43
fs       0x0      0      gs       0x0      0

0x80490e8 <_start>    mov     ebx,0x3
0x80490ed <_start+5>   mov     eax,0x2
0x80490f2 <_start+10>  add     ebx,eax
0x80490f4 <_start+12>  mov     ecx,0x4
0x80490f9 <_start+17>  mul     ecx
>0x80490fb <_start+19> add     ebx,0x5
0x80490fe <_start+22>  mov     edi,ebx
0x8049100 <_start+24>  mov     eax,0x804a000
0x8049105 <_start+29>  call    0x804900f <sprint>
0x804910a <_start+34>  mov     eax,edi
0x804910c <_start+36>  call    0x8049086 <iprintLF>
0x8049111 <_start+41>  call    0x80490db <quit>
0x8049116          add     BYTE PTR [eax],al
0x8049118          add     BYTE PTR [eax],al
0x804911a          add     BYTE PTR [eax],al

native process 33249 (asm) In: _start L13 PC: 0x80490fb
Breakpoint 1, _start () at fix.asm:8
(gdb) si
(gdb) set disassembly-flavor intel
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/dianosov/study/arch-pc/lab09/fix

Breakpoint 1, _start () at fix.asm:8
(gdb) clear
Deleted breakpoint 1
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) |
```

Рис. 4.4: GDB с *fix.asm*

Видим, что на 2 умножается не сумма $3 + 2$, лежащая в регистре `ebx`, а только число 2 в регистре `eax`. Это происходит из-за того, что инструкция `mul` умножает значение регистра `eax` на значение `x`. Чтобы исправить ошибку, поменяем инструкцию `add ebx, eax` на `add eax, ebx`, чтобы значение $3 + 2$ помещалось не в `ebx`, а не в `eax`. Также, поскольку теперь мы имеем результат в регистре `eax`, поменяем `add ebx, 5` на `add eax, 5` и `mov edi, ebx` на `mov edi, eax`.

Изменив код, проверим корректность его работы (рис. 4.5).

```
[dianosov@arch lab09]$ vim fix.asm
[dianosov@arch lab09]$ nasm -f elf -g -l fix.lst fix.asm
[dianosov@arch lab09]$ ld -m elf_i386 -o fix fix.o
[dianosov@arch lab09]$ ./fix
Результат: 25
[dianosov@arch lab09]$
```

Рис. 4.5: Тестирование исправленной программы *fix.asm*

Программа работает корректно. Задание выполнено.

Загрузим файлы на GitHub (рис. 4.6).

```
[dianosov@arch arch-pc]$ git add .
[dianosov@arch arch-pc]$ git commit -am "add files for lab09"
On branch master
nothing to commit, working tree clean
[dianosov@arch arch-pc]$ git push origin master
Enumerating objects: 82, done.
Counting objects: 100% (82/82), done.
Delta compression using up to 12 threads
Compressing objects: 100% (76/76), done.
Writing objects: 100% (76/76), 2.93 MiB | 1.44 MiB/s, done.
Total 76 (delta 26), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (26/26), completed with 4 local objects.
To https://github.com/exterminateddd/pc-course-2024-2025
  6c81bda..fd760d6  master -> master
[dianosov@arch arch-pc]$
```

Рис. 4.6: Загрузка файлов на GitHub

5 Выводы

Приобретены навыки написания программ с использованием подпрограмм.
Освоены методы отладки при помощи GDB и его основные возможности.